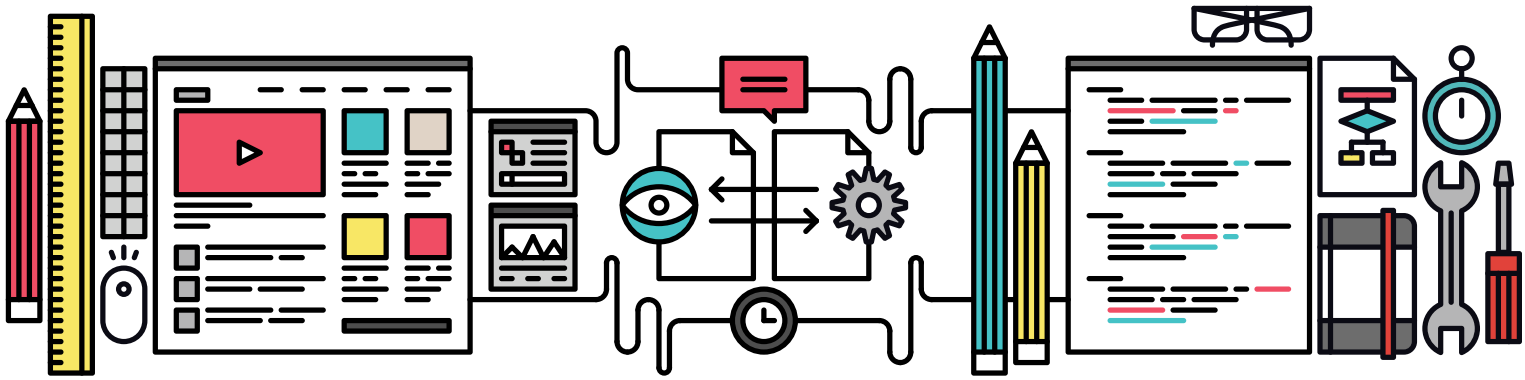
SEPTEMBER 2016

intercom

THE MAGAZINE OF THE SOCIETY FOR TECHNICAL COMMUNICATION



TOOLS OF THE TRADE

WHAT'S THE USE CASE FOR JEKYLL? 6

COLLABORATING AND CONTRIBUTING IN GITHUB 12

GIT HAPPENS 16

GENERATING API DOCS AUTOMATICALLY FROM THE SOURCE CODE 18

CREATING AUTOMATION TOOLS FOR WRITING TEAMS: A PROGRAMMER'S PERSPECTIVE 24

USING DATA-DRIVEN SYNTHESIS TOOLS TO AUTOMATICALLY GENERATE CONTENT 28

AN INTRODUCTION TO SPHINX AND READ THE DOCS FOR TECHNICAL WRITERS 34

TECHNICAL WRITING SUPPORTED BY A PRODUCT LIFECYCLE MANAGEMENT SYSTEM 40

THE BENEFITS OF DITA AUTHORING WITHIN A COMPONENT CONTENT MANAGEMENT SYSTEM 42

Introducing

FLARE™ 12

Create Stunning Web and Print-Based Documentation Your Users Will Love

“One of my favorite new features simplifies creating responsive content. It allows users to select and design a responsive layout medium using the new user-friendly Responsive Layout Editor.”

Denise Kadilak | **Information Architect and Team Manager, Blackbaud**

MAJOR NEW FEATURES

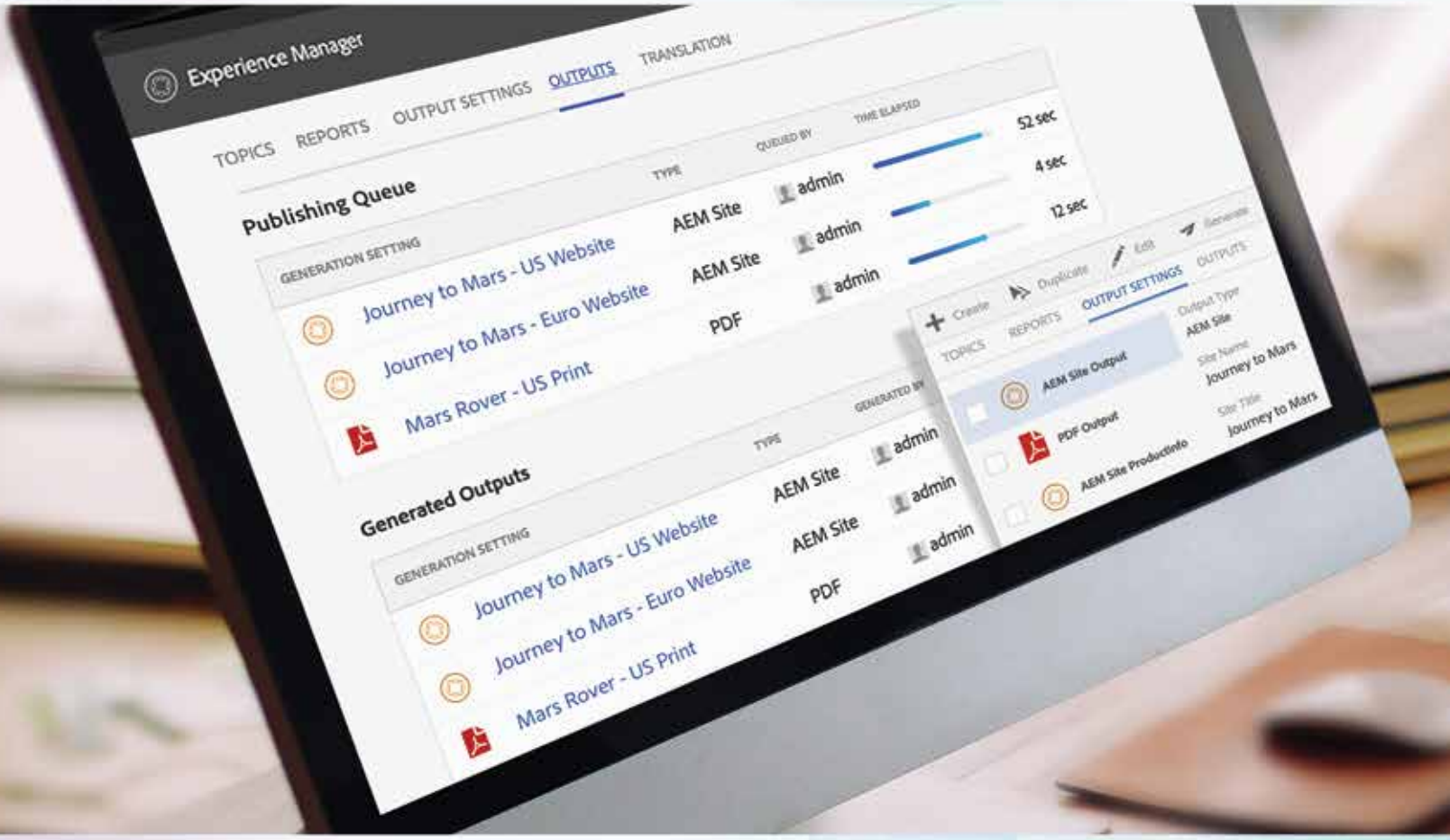
- **Responsive Layout Editor**
  Create Responsive Content without Any Coding or Web Developer Resources
- **New Stylesheet Editor**
  Multiple Medium Views Such as Print, Mobile, Tablet or Any Customized View
- **Multilingual Publishing**
  Multilingual Web and Print Publishing from a Single Project
- **Preserve Tracked Changes**
  Tracked Changes Now Preserved in Word and PDF Output
- **Plus 64-Bit Support, Source Control Improvements, Snippet Enhancements and More!**

Learn More at MadCapSoftware.com/Flare12

Copyright © 2016, MadCap Software, Inc., and its licensors. All rights reserved. MadCap Software, the MadCap Software logo, and MadCap Flare are trademarks or registered trademarks of MadCap Software, Inc., in the United States and/or other countries. Other marks are the properties of their respective owners.

Introducing

XML Documentation Add-on for Adobe Experience Manager



An end-to-end enterprise-class component content management solution for DITA-based content creation and delivery

- Reduce the total cost of ownership by up to 40% by eliminating the need to manage multiple CMSs
- Increase enterprise content velocity by up to three times
- Enjoy all core CCMS functions – such as collaboration, review, approval, translation, search and reports

To request a demo please visit: **http://bit.ly/XMLDocumentationAdd-on**

For any questions, please reach us at techcomm@adobe.com

© 2016 Adobe Systems Incorporated. All rights reserved

September 2016

Volume 63 | Issue 8

5 **From the Editor**
Statement of Ownership

FEATURES

TOOLS OF THE TRADE

6 **What's the Use Case for Jekyll?**
By Tom Johnson
The author argues that Jekyll, a free, open-source tool, is a potential replacement for expensive, complex authoring and publishing systems and help authoring tools.

12 **Collaborating and Contributing in GitHub**
Nicky Bleiel
Learn how to collaborate and contribute in GitHub, including commenting on, reviewing, and merging proposed changes ("pull requests") and managing the wiki community.

16 **Git Happens**
By Morgan Hancock
Tools and processes for developers don't always work for technical writers, but the author explains how Git has been used successfully with her team.

18 **Generating API Docs Automatically from the Source Code**
By Ed Marshall
This article introduces three tools commonly used to generate reference documentation from the source code—Doxygen, Javadoc, and Swagger.

24 **Creating Automation Tools for Writing Teams: A Programmer's Perspective**
By Robert Delwood
In three case studies, the author describes how he improved writing efficiency through automation tools.

28 **Using Data-Driven Synthesis Tools to Automatically Generate Content**
By Joel Kline and Frank Guerino
Data-driven synthesis (DDS) ingest data and automatically generate very large volumes of content that allows practitioners to author content faster, with much higher levels of quality, and for a fraction of the cost.

34 **An Introduction to Sphinx and Read the Docs for Technical Writers**
By Eric Holscher
An overview of the features of Sphinx and Read the Docs, enabling you to evaluate them for use in your organization.

CERTIFIED PROFESSIONAL

Technical Communicator

CPTC™

FOUNDATION

- **Project Planning**
- **Project Analysis**
- **Content Development**
- **Organizational Design**
- **Written Communication**
- **Reviewing & Editing**
- **Visual Communication**
- **Content Management**
- **Production & Delivery**

Certified Professional Technical Communicator™ (CPTC)

Advance your Career and the Profession
Earn the CPTC Foundation Credential Today

The Certified Professional Technical Communicator (CPTC) credential assures employers, colleagues, and the public that you have the knowledge and skill to handle complex technical communication projects from the project planning stage through production and delivery.

Benefits

Why earn the CPTC credential? Because the CPTC credential:

- Distinguishes you from your peers;
- Shows you have the most up-to-date knowledge in the field;
- Opens up job opportunities and enhances job mobility;
- Elevates the profession;
- Gives you a sense of pride in your career; and
- Demonstrates your commitment to the field.

The CPTC credential gives you the edge you need to stand out to employers.

Continuing Education Requirements

Points may be obtained the following ways:

Event	Points
STC Annual Membership (any membership type for Foundation certificants)	2
STC Recorded Webinar (self-study)	1
STC Live Educational Webinar (free, sponsored, and community webinars excluded)	2
STC Online Courses	6
STC Summit Pre-Conference Courses (full day)	6
STC Summit Pre-Conference Courses (half day)	3
STC Virtual Summit	4
STC Annual Summit	8
Begin and complete a college-accredited course related to the Technical Communication field	8
Published articles that relate to any aspect of Technical Communication (2/article)	2
Published books publicly available on topics related to Technical Communication (5/book)	5
Presentations at conferences related to aspects of Technical Communication (2/presentation)	2
Total needed within 2 years post-certification date	**12**

Fees

Exam fees: STC Members $250, Non-Members, $495

Be a leader. ***Take your career to the next level by obtaining your credential. It's the most efficient way to prove your skills and knowledge in the technical communication field.***

CONTACT

For more information about certification and to start the process, visit *www.stc.org* or email *stc@stc.org.*

Advance your Career and the Profession — Earn the CPTC Foundation Credential Today!
Learn more at www.stc.org/certification

40 **Technical Writing Supported by a Product Lifecycle Management System: A Better Approach for the Creation of Product Documentation**
By Christian Brand
Learn the concept of PLM-supported authoring and single-source publishing, as well as the benefits compared to a stand-alone authoring solution or an isolated content management system.

42 **The Benefits of DITA Authoring within a Component Content Management System**
By Keith Schengili-Roberts
A survey of the features and benefits gained when moving your DITA documentation processes from a file system to a CCMS.

ONLINE ONLY

Why Should Technical Writers Embrace Content Analytics?
By Sara Feldman
Learn the benefits and basics of content analytics.

Augmented Reality: Design Techniques and Challenges
By Tena Pair
Technical communicators who want to design augmented reality applications are faced with new design considerations.

How to Buy a Tool
By Liz Fraley
The author recommends strategies for gathering requirements and analyzing tools.

SOCIETY PAGES

44 **2017 Membership Season Now Open**

44 **Reminder of Deadlines for Awards and Honors**

44 **Share Yourself with *Intercom* Readers**

44 **STC Communities and Staff Win Apex Awards**

COLUMNS

THE ESSENTIALS
45 **Handy GitHub Resources**
By Nicky Bleiel

METRICS
46 **Reducing the Cost of Compliance with Intelligent Content**
By Mark Lewis

STANDARD DEVIATION
49 **TechComm Standards: Why They Matter, and Where They Don't**
By Andy McDonald

DEPARTMENTS

FYI
51 **Mark Your Calendar**
Organization Events Across the Globe

OFF HOURS
52 **Do You Want to See My Monkey?**
By Brenda Huettner

ADVERTISERS

C2 **Adobe**
11 **Group Wellesley, Inc**
49 **Hansemeug**
C4 **MadCap Software**
50 **Mercer University School of Engineering**
2 **STC Certification**
4 **STC Education**
14, 27 **STC Membership**
C3 **STC Summit**
41 **XPLM Solution, Inc.**

Experience Professional Growth with STC Education!

STC offers a wide variety of online education options throughout the year. Whether you need an introduction to a subject, an in-depth review, or just a brush-up, STC has what you need. Advance your career with STC's varied collection of online education.

Live Weekly Webinars

Multi-Week Online Courses

Recorded Webinars

Free On-Demand Seminars
(Members Only)

Visit stc.org/education

A Note from the Editor

THIS IS A JAM-PACKED ISSUE of *Intercom*. A dozen authors have written articles on tool-based topics—tools and workflows they are currently using within their technical communication jobs. Popular tools discussed include Jekyll, GitHub, Doxygen, Javadoc, Swagger, Sphinx, and Read the Docs as well as APIs and automation and data synthesis tools. And three articles in the issue have also been posted to *Intercom* Online; those topics include content analytics, augmented reality, and how to evaluate and buy tools. Thank you to Tom Johnson, Nicky Bleiel, Morgan Hancock, Ed Marshall, Robert Delwood, Joel Kline, Frank Guerino, Eric Holscher, Christian Brand, Keith Schengili-Roberts, Sara Feldman, Tena Pair, and Liz Fraley for the outstanding content you contributed and the expertise you shared in this exciting and impressive issue. And thanks also to regular columnists Nicky Bleiel and Mark Lewis and guest columnist Andy McDonald for your expert advice on GitHub resources, intelligent content, and techcomm standards, respectively. I trust technical communicators around the world will find much material in this issue that applies to their work.

September also welcomes a new *Intercom* Editorial Advisory Panel for a two-year term. I met in August with panelists Saul Carliner, Michelle Despres, Linda Oestrieich, Marta Rauch, Kelly Schrank, and Kirk St.Amant to discuss the 2017 editorial calendar. Stay tuned for new themes for 2017 to be published in the October issue and, as always, feel free to send me your submission and topic ideas as well as your feedback.

—Liz Pohland
liz.pohland@stc.org

STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION

INTERCOM (ISSN 0164-6206) s published ten times a year by the Society for Technical Communication, a nonprofit educational organization, as a service to the membership. Members may pay $60 for a print subscription to INTERCOM.

The mailing address of both the publication and the publisher is 9401 Lee Highway, Suite 300, Fairfax VA 22031-1803. The publisher is Elizabeth Pohland, and the editor is Elizabeth Pohland. The owner of the publication is the Society for Technical Communication.

There were 581 copies of INTERCOM published in July/August 2016. The average for the preceding 12 months was 538. The paid/requested outside-county mail subscriptions for the July/August 2016 issue were 531; the average for the preceding 12 months was 466. Total distribution for July/August 2016 was 531. The average for the preceding 12 months was 466. Ten free copies of INTERCOM were distributed by mail in July/August 2016 and the average number of free copies distributed during the preceding 12 months was 15. Forty copies of INTERCOM were not distributed in July/August 2016, and the average number of copies not distributed during the preceding 12 months was 50. The percent paid/requested circulation in July/August 2016 was 92 percent; for the preceding 12 months, percent paid/requested circulation was 87 percent.

intercom

Intercom, the magazine of the Society for Technical Communication, is published to provide practical examples and applications of technical communication that will promote its readers' professional development.

Publisher Society for Technical Communication
Editor Elizabeth E. (Liz) Pohland
Assistant Editor James Cameron

CONTRIBUTING EDITORS
Scott Abel, *Meet the Change Agents*
Thomas Barker, *The Academic Conversation*
Nicky Bleiel, *The Essentials*
Kit Brown-Hoekstra and Cindy Currie, *Ask a Tech Comm Manager*
Jeanette Evans and Charles Dull, *Emerging Technologies*
Ray Gallon, *Standard Deviation*
Mark Lewis, *Metrics*
Linda Roberts and Lisa Cook, *All Access*
Derek Ross, *Ethics*
Kirk St.Amant, *Student Perspectives*
Kirk St.Amant, *Working Locally, Communicating Globally*
Brian Still, *Trends in Usability*

EDITORIAL ADVISORY PANEL
Saul Carliner (Chair), Michelle Despres, Linda Oestreich, Marta Rauch, Kelly Schrank, and Kirk St.Amant

STC BOARD OF DIRECTORS
Officers
Adriane Hunt, *President*
Alyssa Fox, *Vice President*
Bernard Aschwanden, *Immediate Past President*
Jane Wilson, *Treasurer*
Kirsty Taylor, *Secretary*
Directors Craig Baehr, Alisa Bonsignore, Liz Herman, Cindy Pao

SOCIETY FOR TECHNICAL COMMUNICATION
9401 Lee Highway, Suite 300
Fairfax, VA 22031-1803
+1(703) 522-4114 | +1(703) 522-2075 fax
Send questions and correspondence to the editor at the above address. General information about STC: stc@stc.org *or* www.stc.org.

Address Changes: membership@stc.org
Intercom Editorial: intercom@stc.org

Reprint Permissions: www.copyright.com
Intercom (ISSN 0164-6206) is published 10 times a year (monthly, except for August and December) by the Society for Technical Communication, 9401 Lee Highway, Suite 300, Fairfax, VA 22031-1803, a nonprofit educational organization, as a service to its membership. Membership in STC includes a subscription to **Intercom**. Periodicals postage paid at Fairfax, VA, and additional offices.
Postmaster Send address changes to **Intercom**, Attn: Membership, c/o Society for Technical Communication, 9401 Lee Highway, Suite 300, Fairfax, VA 22031-1803.

Copyright © 2016 by the Society for Technical Communication. All rights reserved. Printed in the USA. All articles in **Intercom** are copyrighted by the authors unless otherwise indicated. Anyone interested in reprints, reproductions, or reuse must contact the authors for permission.

To advertise in **Intercom**, contact Stacey O'Donnell at *stacey.odonnell@stc.org*, Society for Technical Communication, 9401 Lee Highway, Suite 300, Fairfax, VA 22031. Ph: +1(703) 522-4114; Fax: +1(703) 522-2075.

shutterstock.com/ dotshock

What's the Use Case for Jekyll?

By TOM JOHNSON | *STC Senior Member*

I RECENTLY GAVE a presentation titled "Writing tech docs like a hacker with Jekyll" at an STC Silicon Valley Chapter meeting, held in a backroom at IHOP around some long tables, with a slide projector positioned up front. "What's the use case for Jekyll?" an attendee asked, repeatedly. No matter how much I tried to explain Jekyll's usefulness, I never quite found an answer she liked.

"Yeah, but I can already do all this in our publishing system," she said. She worked for one of the big tech companies and had an extensive DITA implementation set up using something like IXIASOFT or Trisoft. Everything Jekyll could offer, her publishing system already provided. So what was really the reason why she or anyone else should be interested in Jekyll?

It's true that if you compare help authoring tools feature by feature, a heavy DITA publishing system or even a popular HAT like Madcap Flare would likely beat Jekyll.

I explained that nearly everything you can do with DITA you can do with Jekyll, but ultimately the question returned, and she said, "I just don't see what the use case for Jekyll is."

Having been in the field for a while, I've used a range of help authoring tools—RoboHelp, Flare, Microsoft Word, Mediawiki, Author-it, Drupal, Google Docs, Confluence, DITA with OxygenXML, and more. With each tool, I've always come up against shortcomings or annoyances with the tools that bothered me. When I had autonomy with tools, I often abandoned one and moved on to the next, only to move again along that familiar hype cycle from the peak of expectation into the trough of disillusionment.

What Is Jekyll?

I should probably back up a bit and introduce Jekyll (*http://jekyllrb.com*), since few technical writers are familiar with it. Jekyll was originally created in 2008 by Thomas Preston-Werner, co-founder of GitHub, as an alternative to CMS-based blogging tools.

Preston-Werner wanted to approach blogging like a developer, using developer tools and workflows. Preston-Werner writes, "On Sunday, October 19th, I sat down in my San Francisco apartment with a glass of apple cider and a clear mind. After a period of reflection, I had an idea. While I'm not specifically trained as an author of prose, I am trained as an author of code. What would happen if I approached blogging from a software development perspective? What would that look like?"

This seminal post helped launch a whole breed of tools known as "static site generators." There are now *hundreds* of static site generators (see *staticgen.com*), but based on user activity, Jekyll is by far the most popular and has the most community support.

jekyll
HOME
DOCUMENTATION
NEWS
HELP
VIEW ON GITHUB
Transform your plain text into static websites and blogs.
Simple
No more databases, comment moderation, or pesky updates to install—just your content.
How Jekyll works →
Static
Markdown (or Textile), Liquid, HTML & CSS go in. Static sites come out ready for deployment.
Jekyll template guide →
Blog-aware
Permalinks, categories, pages, posts, and custom layouts are all first-class citizens here.
Migrate your blog →
Get up and running in seconds.
Quick-start instructions
~ $ gem install jekyll
~ $ jekyll new my-awesome-site
~ $ cd my-awesome-site
~/my-awesome-site $ jekyll serve
=> Now browse to http://localhost:4000

Figure 1. Jekyll has more than twice as many stars and four times as many forks as the next competing static site generator, GitBook.

As a static site generator, Jekyll takes all your Markdown files and generates them into a complete HTML website. That's where the name Jekyll comes from—the ability to *transform* something that's plain into something awesome. There's a bit more going on here besides rendering Markdown, though.

Jekyll Project Setup

With Jekyll, you store all your files in a project folder. Basic settings are defined in a configuration file. Using a terminal emulator (I prefer iTerm), you change to your Jekyll project folder and then run "`jekyll serve`" to create a preview of your content on a Web server.

I usually keep this preview window open in one browser and my editor (I like WebStorm) open next to it. As I work

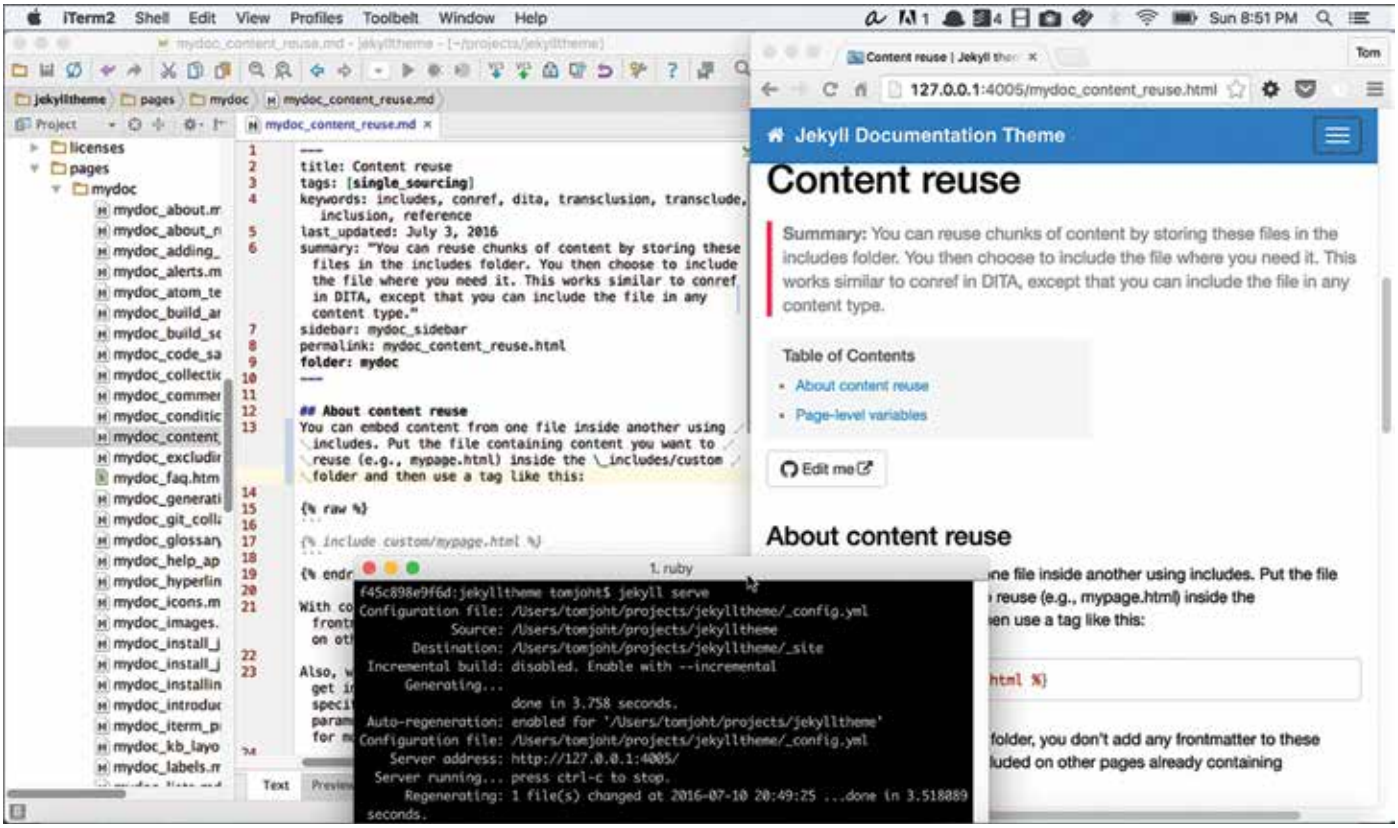

Jekyll Documentation Theme
Content reuse
Summary: You can reuse chunks of content by storing these files in the includes folder. You then choose to include the file where you need it. This works similar to conref in DITA, except that you can include the file in any content type.
Table of Contents
About content reuse
Page-level variables
Edit me
About content reuse
127.0.0.1:4005/mydoc_content_reuse.html
Configuration file: /Users/tomjoht/projects/jekylltheme/_config.yml
Source: /Users/tomjoht/projects/jekylltheme
Destination: /Users/tomjoht/projects/jekylltheme/_site
Incremental build: disabled. Enable with --incremental
Generating...
done in 3.758 seconds.
Auto-regeneration: enabled for '/Users/tomjoht/projects/jekylltheme'
Server address: http://127.0.0.1:4005/
Server running... press ctrl-c to stop.
Regenerating: 1 file(s) changed at 2016-07-10 20:49:25 ...done in 3.518889 seconds.

*Figure 2. My text editor is open on the left; the browser on the right is open to the preview URL (*http://127.0.0:4005*). The terminal window shows how the site regenerates each time I save a change.*

on content and save changes, Jekyll rebuilds the site and shows me the update. This way I can work in a text editor while also seeing what the output will look like.

Advanced Scripting Logic

Beyond just converting Markdown, Jekyll also uses Liquid (*https://help.shopify.com/themes/liquid*), a scripting language created by Shopify, to implement more advanced logic with your content. Liquid includes scripts that can run "for" loops through your content, process "if" and "elsif" statements, apply filters to strings, and more. For example, you might define a property in your configuration file called `audience`. In one output, you could set `audience` equal to `administrators`. In another, you could set it equal to `analysts`. Then you could incorporate logic on a page like this:

```
{% if site.audience == "administrators" %}
show this to administrators
{% elsif site.audience == "analysts" %}
show this to analysts
{% endif %}
```

The outputs would conditionalize the content to show the correct content to the right audience. You can also store content in an "includes" folder. To embed any file from your includes folder into another file, you use an include tag like this:

```
{% include myfile.md %}
```

The content from myfile.md gets inserted where you placed the tag. I use includes not only for content re-use, but also to create templates for alerts (notes, tips, cautions) as well as for images. You can pass parameters into the template and populate it in easy ways.

This is only the tip of the iceberg when it comes to Liquid and Jekyll. When you see what you can do using Liquid, it usually removes fears people have about Markdown being too primitive and limiting to handle robust use cases common with tech docs. In addition to Markdown and Liquid, you can also use any HTML or JavaScript directly in your pages.

No Database Backend

Static site generators contrast sharply with Web-based content management systems like WordPress or Drupal. With Web-based content management systems, when users go to your page, PHP tags make calls to the database for the page's content. The content gets served up from the database and dynamically inserted into the page for the user to view. This rendering takes time.

With Jekyll and other static site generators, however, content is already populated onto the page. When users go to the page, the page content immediately appears, because there's no need to make a trip to a database to retrieve the content. As a result, Jekyll pages load noticeably faster (half a second instead of the two-second average with Web CMSs).

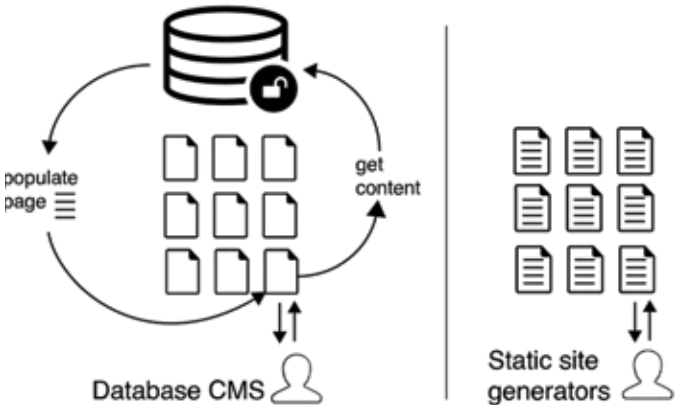


Figure 3. Web-based CMSs pull content from a database every time a user visits a page. Static site generators don't have a backend database; content is already pre-packaged onto the page.

Additionally, because there isn't a database or online login screen, Jekyll sites are much more secure. There's nothing to hack. You won't run into any software restrictions from your security group.

Integration with Version Control

Usually you use version control (like Git or Mercurial) with your Jekyll project. Additionally, GitHub offers some slick features to support Jekyll. When you commit your Jekyll source files to a GitHub repository, in a branch called "github-pages," GitHub will automatically build your Jekyll site for you. This feature is called GitHub Pages (*https://pages.github.com*).

A couple of years ago, I converted my WordPress blog to Jekyll (building with GitHub Pages) and have been pleased with the workflow. Services similar to GitHub Pages—such as a plugin called "Jekyll-hook"—are available for you to implement on your own server.

GitHub Pages
Websites for you and your projects.
Hosted directly from your GitHub repository. Just edit, push, and your changes are live.
jekyll
Transform your plain
static websites and b
Simple
Static
Transform your plain text into static websites and blogs.
Simple

Figure 4. GitHub provides hooks to automatically build a site from your repo commits when the source is a Jekyll project.

Markdown and Text-file Formats

Working with Jekyll for the past two years, I've found that I really like working in Markdown and text file formats. I cringe every time I open a help authoring tool and see a WYSIWYG editor. I also like the freedom to implement any layout I want using basic HTML, CSS, and JavaScript.

For example, I recently wanted to embed a linear workflow map at the top of each topic in a series, as seen in Figure 5.

Jekyll Documentation Theme
News
Jekyll Resources
Products
Feedback
Product2 1.0
Introduction
Simple Workflow
Sample 1
Sample 2
Sample 3
Sample 4
Sample 5
Complex Workflow
Sample 1 Topic
Connect to ADB
Download and Build the Starter Kit
Take a Tour
Load Your Widgets
Query for Something
Summary: This is just a sample topic...
Table of Contents
Sample Content
More sample content
Sample Content
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.
It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here,

Figure 5. This workflow map appears at the top of a lengthy procedure. Jekyll made it simple to implement this map using basic HTML, CSS, and JavaScript.

The workflow squares, which are drawn with CSS, automatically highlight based on a box number stored in the page's frontmatter. Some JavaScript looks at the box number and assigns an `active` class to the box to perform the green highlighting.

Jekyll is incredibly flexible and allows me to configure outputs for nearly any documentation scenario.

Here's another example. To deliver context-sensitive help for a project, I used collections in Jekyll to generate a JSON file that Web engineers integrated into a software application. Code on each application page simply pulled in the tooltip content from the JSON file I supplied. (I could have even hosted the JSON file on a separate help server, making it act like a content API, but it was easier to just include the JSON file with the application.)

Basically, you can do pretty much anything you want in Jekyll—especially if you have some Web savvy. Even if you don't have Web savvy, Jekyll's code is surprisingly simple (definitely simpler than WordPress). Building a website to match your corporate brand involves just a few basic steps:

1. Copy the source of the site.
2. Make all the JS and CSS links work locally.
3. Gut the page content and replace it with a `{{content}}` tag.
4. Save this code as mylayout.html in the layouts folder of your project.
5. In your Jekyll page frontmatter, specify `layout: mylayout.html` for your page.

Ultimately, the use case for Jekyll is that it's a replacement for help authoring tools. If HTML is your main output, you can probably do most everything with … Jekyll.

When Jekyll builds the site, your Markdown page content will be inserted into the `{{content}}` tag of the layout you just defined. That's it. You don't have to sort through a long list of PHP tags, there's no database to configure, no complicated templates with loops to set up. Jekyll is easy enough for technical writers to extend and customize for their unique authoring and publishing needs.

Ultimately, the use case for Jekyll is that it's a replacement for help authoring tools. You don't need to spend thousands or hundreds of thousands of dollars each year paying vendors to build elaborate systems to handle authoring and publishing. If HTML is your main output, you can probably do most everything with an open source, static site generator like Jekyll.

With Jekyll, you can handle robust re-use, conditional filtering, variables, multiple outputs, and even—with a bit of hacking—PDF output using a utility called Prince XML.

So my question to the attendee was the same as the one she posed to me, but in reverse, "What's the use case for these expensive, complex authoring and publishing systems when I can do all of this with a free, open-source tool like Jekyll?" Additionally, Jekyll gives you some unique advantages:

- Complete control over the site output
- The ability to work in Markdown formats
- Integration with version control
- Lightning fast loading times
- No licensing costs

A Few Limitations

Jekyll does have some limitations. Jekyll won't tell you if any links are broken. You can't see which topics link to each other. While PDF output is possible with some hacks, Jekyll wasn't designed to easily generate PDF. Because Jekyll is open, it's hard to lock down content to a specific set of allowed tags or properties.

If you're using Jekyll with version control, you'll need to become familiar with tools like Git or Mercurial, which might require additional learning. Search isn't provided out of the box (you have to incorporate third-party search such as Google, Algolia, or Swiftype). Nor are comment forms provided (again, you would need third-party services such as Disqus). Users can't log in and have profiles, which means you can't restrict access based on user roles.

Despite these limitations, Jekyll seems to have hit the sweet spot with how I like to operate. It's a tool I don't seem to grow out of. The more I immerse myself in developer workflows and environments, the more comfortable I become using Jekyll and working in a text editor and terminal.

Jekyll probably makes the most sense in developer documentation environments, especially when developers might be contributing or working with documentation. If developers write or edit any docs, they usually prefer to write in Markdown instead of XML.

Additionally, Jekyll works extremely well with API documentation, as it provides powerful syntax highlighting and lets you easily include code samples (either embedded directly or referenced from another directory).

My Documentation Theme for Jekyll

If you're interested in exploring Jekyll, check out the "Documentation theme for Jekyll" that I've created (*http://idratherbewriting.com/documentation-theme-jekyll*) document. This theme is more than just a pretty skin for your content. In this theme I try to outline best practices and techniques for all the common scenarios that technical writers face.

From content re-use to alerts, automated links, robust table of contents, conditional filters, context-sensitive help, and even PDF output, I outline my approach and show how I implemented it in the Jekyll theme. It can also be helpful to look at some other documentation sites that use Jekyll:

- Bootstrap (*http://getbootstrap.com/*)
- Beegit (*http://help.beegit.com/*)

- Stack Overflow blog (*https://blog.stackoverflow.com/*)
- RethinkDB (*http://rethinkdb.com/docs/*)
- Github docs (*https://help.github.com/*)
- Basekit (*http://docs.basekit.com/*)
- Jekyllrb docs (*http://jekyllrb.com/docs/home/*)
- SendGrid docs (*https://sendgrid.com/docs*)
- Atlassian Design (*https://design.atlassian.com/*)
- Cloud Cannon docs (*https://docs.cloudcannon.com/*)
- Wistia help center (*http://wistia.com/support*)
- devo.ps documentation (*http://docs.devo.ps/*)

Sometimes looking at source code of other projects can give you ideas for how to approach similar issues. (Most of these projects have open GitHub repos.) This openness leads to more collaboration and innovation, with multiple communities building on each other instead of competing against each other.

Jekyll probably makes the most sense in developer documentation environments, especially when developers might be contributing or working with documentation.

TOM JOHNSON *(tom@idratherbewriting.com) is a technical writer at Amazon Lab126 (*www.lab126.com*) in Sunnyvale, California. He is most well-known in the community for his tech comm blog at* http://idratherbewriting.com. *You can access the Jekyll documentation theme and other resources from the Jekyll tab on Tom's site. You can contact Tom through Twitter* @tomjohnson.

REFERENCES

Jekyll versus DITA series, *http://idratherbewriting.com/2015/03/23/new-series-jekyll-versus-dita/*

Preston-Werner, Thomas. 17 Nov. 2008. Blogging like a hacker, *http://tom.preston-werner.com/2008/11/17/blogging-like-a-hacker.html*

CERTIFIED PROFESSIONAL
Technical Communicator
CPTC™
FOUNDATION

CPTC Foundation-Level Training

Washington DC: October 20-21

Toronto: November 21-22

Other cities and dates to be announced!

Our guarantee: If you do not pass the CPTC Foundation exam on your first attempt after completing the Group Wellesley CPTC training, *we will pay 50% of the cost of your retest.*

For details on all upcoming public courses or to schedule a virtual or private course, contact training@groupwellesley.com, or visit www.groupwellesley.com/cptc.

gw GROUP WELLESLEY, INC.

Collaborating and Contributing in GitHub

By NICKY BLEIEL | *Associate Fellow*

shutterstock.com/ venimo

GITHUB (*http://github.com*) is a Web-based repository for software projects and is reportedly the world's largest open source community, hosting over 35 million repositories that include both code and the documentation for that code. It includes version control, project management, and social features that technical communicators can use to manage and document software projects.

Technical communicators can collaborate and contribute in GitHub in a variety of ways, including: writing content and managing issues; commenting on, reviewing, and merging proposed changes; documenting GitHub projects; and managing the wiki community.

GitHub is also an excellent choice for version control of documentation files and management of documentation issues. Docs aren't siloed in GitHub—they live with the code, follow the same workflow as the code, and are reviewed with the code—which is especially useful in an Agile environment, where docs are part of the "definition of done."

GitHub projects can be public, private, or hosted internally behind your company firewall on GitHub Enterprise. While the features and options are the same for all, your responsibilities can vary for each. Company projects hosted on public GitHub need customer-facing documentation and possibly community monitoring. Projects hosted behind a firewall in GitHub Enterprise need internal documentation, issue review and tracking, and project file management. When contributing to public GitHub projects, you can pick and choose what you want to do—write or edit the documentation, or contribute code, graphics, or comments.

It All Starts with Git

GitHub uses Git as a repository. Git (a command-line tool initially released in 2005) is a distributed, nonlinear version control system for software development by Linus Torvalds, creator of Linux. Git has wide adoption; according to a survey released in May 2014 by the Eclipse Foundation, Git is being used by one third of software developers and is the #1 code management tool.

Why GitHub?

GitHub could be considered a "one-stop-development-shop" because, in addition to version control, it includes issue tracking (bugs and feature requests), notifications, diffs, status dashboards, and documentation. It also has social features. You can "follow" other contributors and GitHub will automatically alert you of their activities. You can also "watch" specific projects—you will receive notices about those also. If you prefer to tag a repository, but skip the notifications, you can "star" it. You can then go to your "stars page" to catch up on those projects, as well as take a look at the "stars" of your friends.

One of the main reasons GitHub has become so popular is that it has simplified the process of contributing to open source projects. According to Gregg Pollack of Code School (in the article "What Exactly Is GitHub Anyway?"), before GitHub, contributing to a project meant a developer had to download the source code, make changes, create a patch, then email that patch to be evaluated. With GitHub, you can create a local copy (a "fork") of the source code, make your changes, and submit a request for consideration (a "pull" request). And there is a public record of all of this, so you can build a reputation in the community.

Organizations can post projects on the public version of GitHub for free, as well as purchase GitHub Enterprise for their internal use behind a firewall. Individuals can create public repositories and contribute to projects with a free account, but must pay a fee to create private repositories that are accessible only to them and their invited collaborators.

GitHub Terminology

Following is a list of selected GitHub terminology from the GitHub Glossary (*https://help.github.com/articles/github-glossary/*):

- **Repository** (also referred to as a "repo"): A repository is the most basic element of GitHub. They're easiest to imagine as a project's folder. A repository contains all of the project files (including documentation), and stores each file's revision history.
- **Fork**: A fork is a personal copy of another user's repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original.
- **Pull request**: Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators.

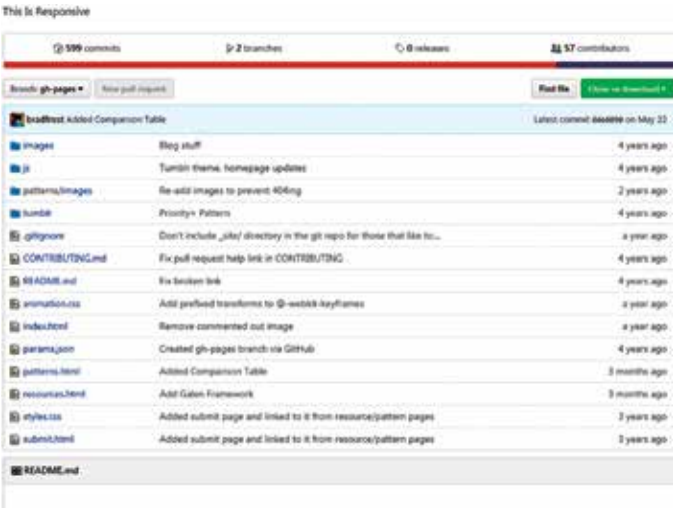


Figure 1. A Sample GitHub Repository

- **Merge**: Merging takes the changes from one branch (in the same repository or from a fork), and applies them into another.
- **Collaborator:** A collaborator is a person with read and write access to a repository who has been invited to contribute by the repository owner.
- **Contributor:** A contributor is someone who has contributed to a project by having a pull request merged but does not have collaborator access.

GitHub Documentation Options

There are three options to document a GitHub project: READMEs, GitHub Pages, and a Wiki. Public GitHub projects may use all three to explain and publicize the project, as well as build community. An internal GitHub Enterprise project could include more than one README (for example, one for development plans and one for notes on final customer-facing docs); the Wiki could document processes (or not be used); or GitHub Pages may not be used at all.

READMEs

When you create a project in GitHub, a README file is created automatically. Unlike most READMEs, GitHub readmes are front and center and are the "hub" of the repository. You can create more than one if you need it, but the one named README.md will be displayed by default when the project is opened.

Figure 2. The Octocat is the official mascot of GitHub (https://github.com/cameronmcfee). *The Ordered Listocat, a variation from the Octodex, is by Cameron McEfee* (https://octodex.github.com/).

GitHub could be considered a "one-stop-development-shop."

READMEs are edited in Markdown, and have the file extension of .md. Markdown is a lightweight markup language that can be converted to HTML easily. If you have used wiki syntax, it will look very familiar. Markdown was originally developed in 2004 by John Gruber and has splintered into different variations. GitHub uses "GitHub Flavored Markdown," which has features such as syntax highlighting, task lists, tables, and @mentions (a quick way to notify someone that you need them to take a look at something). Markdown is also used for GitHub Wikis, GitHub Pages, for commenting, and when creating sharable reusable snippets, called Gists.

In addition to contributing content to the readme, any proposed changes need to be reviewed, commented on, and merged. You can also review the READMEs in public repositories and propose changes to them. (This README checklist on GitHub by Daniel Beck is a useful guide to writing quality READMEs: *https://github.com/ddbeck/readme-checklist/blob/master/checklist.md.*)

Wikis

You can expand upon the information in the README by adding wiki pages to your project repository. Wiki pages are easy to create, and can be edited in Markdown or one of the other eight edit modes. By default, anyone can edit a GitHub project wiki, but you can change the settings and make your wiki read-only. If your project wiki is public, contributions from the community will need to be monitored. To learn how to setup a wiki in GitHub, see the Mastering Wikis tutorial in GitHub Guides at *https://guides.github.com/features/wikis/.* (Examples of GitHub wikis can be found online at *https://github.com/showcases/projects-with-great-wikis.*)

GitHub Pages

Another way to provide documentation for your project is to use GitHub Pages. These are Web pages hosted and published on GitHub. They are authored in Markdown, and you can use GitHub-provided themes to create a custom look. You can even add your Google Analytics tracking ID to each of your Pages. After you are satisfied with your content and the theme, you can publish your GitHub Pages and the default URL will be: *http://[accountName].github.io/[repoName].* GitHub Pages are always public, even if your repository is private. (Examples of GitHub Pages can be found online at *https://github.com/showcases/github-pages-examples.*)

Using GitHub for Version Control

GitHub has, at its core, the Git version control system. Files are stored in Git repositories, and Git is a powerful tool. If you have used Git for version control before, that knowledge will be useful when working in GitHub. (A quick reference to Git commands can be found online at *https://git-scm.com/docs.*)

If your company is using GitHub Enterprise, it makes sense to use it for the version control of your documentation files. And storing documentation files along with code and other project artifacts means that docs are not siloed, can follow the same workflow as code, and that the project's collaborators can review and contribute.

There are some things you need to keep in mind when using version control in GitHub. First, you need to determine what workflow you will use. In public GitHub, it is fairly straightforward: you "fork" (make a copy on your account) of the project, clone or download a local copy of the project to your machine, make your changes, commit ("push") them on your fork, and then submit a request for consideration (a "pull request"). The project owners and the community can comment on the changes, and the owners can merge them if they approve. You can do all of this in the GitHub graphical user interface (GUI), but you can also work strictly on the command-line using Git commands, or with a mixture of both.

In GitHub Enterprise, the workflow will be similar, but since you will be a collaborator, you can upload the documentation files to the "master" branch, and create new branches to make your revisions in. (In GitHub, the "master" branch is defined as the one that can be

One of the main reasons GitHub has become so popular is that it has simplified the process of contributing to open source projects.

released at any given time, but could also be considered the "original" version that you are refining incrementally on branches.) Your company may develop guidelines you need to follow around the naming of branches and who should do reviews and merges. You should be involved in the development, and, of course, the documentation of these guidelines. (See *https://help.github.com/articles/what-is-a-good-git-workflow/* for an overview of Git workflows.)

In GitHub, any file that can be read with a text editor can be opened and edited. When working with Markdown files and many other file types, editing and reviewing can be done right in the GitHub GUI. But for some files (like .dita files), you may prefer to use your XML editing tool. And binary files (images, Word files, etc.) can't be opened within GitHub at all. This is part of the reason why working on a local copy ("clone") of your GitHub project is a best practice. On your local copy, you can open any file in the application you wish.

Managing Documentation Issues in GitHub

You can manage documentation issues (bug and feature requests) in your project using GitHub's issue tracking. If you have used other issue tracking systems, many of the features will be familiar, and as with any system, you need to think about how granular issues should be, how you want to tag them, and what best practices your team should follow. GitHub issues can be authored in Markdown, so you can add formatting, create task lists to track progress, use emojis, and more.

It is important to create all the Labels (such as "Documentation") and Milestones you need so that you can tag the issues in your project properly. Every issue should be tagged with one or more labels, a milestone, and an assignee (owner). It is easy to find issues using the Filters and Search, and you can click on any label or milestone to find all issues in a project that contain those tags. You can also search for issues across repositories, see *https://help.github.com/articles/searching-issues/* for details. If you need to break a larger documentation project down into subprojects, you can create Epics and assign issues to them. To reference another issue in your project, enter "#" in the description or a comment and the entire list of issues for that project will display. The chosen issue will become a link you can use to navigate between the issues.

After an issue is created, anyone with Collaborator status in that project can comment on it. If you would like a specific team member to comment on an issue, use an @ mention in the issue description or a comment. As soon as you enter "@", the list of collaborators will display and you can choose one.

A fun GitHub convention you may want to adopt is including the "Ship It Squirrel" emoji (:shipit:) in a comment if you believe the proposed change is ready to ship.

To view and manage your issues on a virtual task board (which is great for agile shops and virtual teams), check out ZenHub, a productivity tool that can be integrated with GitHub. (More information about GitHub issues can be found in the Mastering Issues tutorial in GitHub Guides online at *https://guides.github.com/features/issues/*.)

Integrations with Other Tools

In addition to ZenHub, over 70 productivity tools can be integrated with GitHub, including Slack, a team messaging and collaboration application. If you would prefer a Git GUI client instead of working at the command line to clone, commit, push, etc., check out GitKraken and GitHub Desktop. GitBook can be used to host, write, and publish documentation (outputs include PDF, ePub, mobi or a website). (See the complete list of tools online at *https://github.com/integrations*.)

Other Uses for GitHub

GitHub's origins are in software collaboration, but it can also be used to collaborate on other types of projects, and is a great place to post your portfolio or class projects. According to the article "From Collaborative Coding to Wedding Invitations: GitHub Is Going Mainstream," it is also being used to collaborate on projects as diverse as Gregorian chants, licensing agreements, and wedding invitations.

NICKY BLEIEL *(*nableiel@us.ibm.com*) is a Watson Information Developer at IBM and a GitHub user. She is a Past President and Associate Fellow of the Society for Technical Communication and has over 20 years of experience writing and designing content for software products in a variety of industries. She is a popular speaker at many conferences, including the STC Summit, WritersUA, tcworld, CIDM, and LavaCon; and has been published in STC's* Intercom, tcworld magazine, ISTC Communicator, *and more. Learn more about her at* nickybleiel.com.

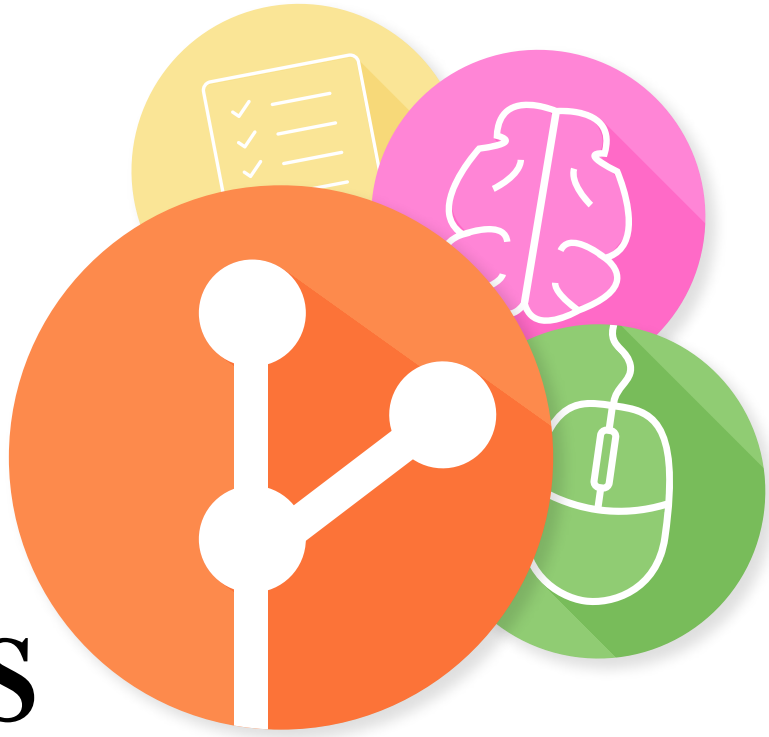
REFERENCES

Finley, Klint. "What Exactly Is GitHub Anyway?" *TechCrunch* (14 July 2012). *http://techcrunch.com/2012/07/14/what-exactly-is-github-anyway/.*

McMillan, R. "From Collaborative Coding to Wedding Invitations: GitHub Is Going Mainstream," Wired (2 September 2013). *www.wired.com/2013/09/github-for-anything/.*

Skeritt, Ian. Eclipse Community Survey 2014 Results (2014). *https://ians.kerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/.*

Git Happens



By MORGAN HANCOCK

I'D BE LYING if I said the potential puns didn't fuel my passion for Git at least a little bit. I titled my speaking proposals "Git Ready!" and my training courses "Git Started!" I'd often catch myself saying, "Do you *git* it?"

Cheesy, low-hanging puns aside, six months ago Git was a foreign source control system to the technical writing team at Bazaarvoice, a business-to-business software company. We were using SVN (Subversion), which we still use for some products. However, the software development teams at Bazaarvoice had long abandoned SVN for Git. Many developers prefer Git because it's a distributed source control system that tracks a repository's history, not just differences between file versions. However, tools and processes for developers don't always work for technical writers, but still I wondered, *would Git?*

Exploring Git

Git is a distributed source control system that tracks a repository's history, not just file changes, but what exactly does that mean?

In SVN, a commit added changes to the repository for specified files. In Git, a commit is a snapshot of the repository. When you view a Git commit, you won't just see the files that changed. You'll see what the entire repository looked that at that point in time. So a Git commit captures changes to files but also records unchanged files.

This method of tracking changes is beneficial to software developers when an error or bug breaks their code because Git allows them to easily roll back their repository to a safe commit. They don't need to revert a ton of individual commits to return stability; they can simply roll their branch back to the last stable commit.

My writing team could benefit from this, too. We recently updated the name of a feature and republished the knowledge base (KB), only to be told the next day that the name change wouldn't take effect for a few weeks. We had to revert each file we touched before we could rebuild. If we had been using Git, we could have simply reverted to the previous, stable commit and republished. We could have also maintained the changes we had made and deployed them when the name change took effect, instead of having to manually apply the changes a second time, doubling our work.

An increasing number of frustrations combined with dwindling SVN support at our company made my team realize we needed to change our source control system. We decided to mirror our developers and migrate to Git.

Workflow

Before we actually moved any files to Git, we needed to identify a workflow that meshed with our processes. In our SVN workflow, we developed content locally, used PDFs, HTML, or MadCap Flare review packages for reviews, and committed changes to the SVN repository when ready to publish. We knew our SVN workflow wouldn't directly translate to Git, so we researched several popular Git workflows. Ultimately, we decided that GitHub Flow best met our needs and would be easy to follow.

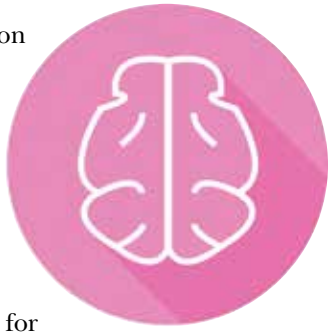
GitHub Flow (now our workflow) relies on a golden rule—the master branch is always deployable. This means that the master branch is always stable and only contains content ready for publication. Publishing from this branch is always safe. We consider everything in the master branch published or about to be published.

To create new documents or edit existing content, a writer branches off of master and develops in that new branch. The writer commits changes locally and pushes that branch and those commits to the remote repository regularly. To collaborate, other writers can work in that branch and share new commits, or create their own branch and submit a pull request to ask the other writer to merge their commits into that development branch.

In addition to our traditional review methods—creating a PDF for review or sharing a MadCap Flare review package—Git gave us additional avenues for reviews and collaboration. We can send SMEs to view changes in a specific commit on GitHub. At the end of our review process, we submit a pull request to merge our development branch into the master branch. Instead of having a traditional lead review, we can ask our lead to view the pull request, review our changes, make comments, give feedback, and accept the pull request if approved or after implementing the new feedback. This streamlines our process quite a bit and has improved efficiency. Some writers prefer the traditional review methods, but others have embraced the GitHub-powered alternatives. Either way, Git has provided additional options that our team can choose when appropriate.

However, before we actually used this new workflow, we had to face the steep uphill battle of learning how Git works and how to use it.

Training



As the most easily excited person on my team, I enthusiastically volunteered to develop Git training for our team. I had no idea what a deep abyss I was about to plunge into.

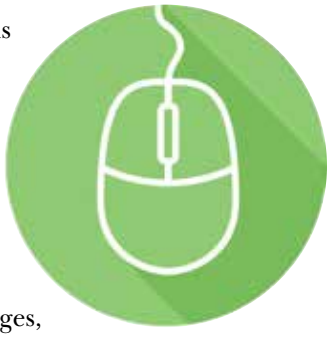
The Internet is overflowing with amazing and detailed resources for Git. But almost all of the helpful resources are for developers and coders, people not like me and definitely not like my team. I couldn't just corral a bunch of links and tell my team to start reading. We'd have been back home with SVN almost instantly. So I had to learn Git for real and filter out the noise before I could train my teammates on it.

Here's the thing about Git—it's a very powerful source control tool for developers. There are so many commands that a developer may need that a technical writer would have no use for. I didn't want to overwhelm my team with *all the knowledge!* So I had to sift through commands and concepts and figure out what my team really needed to know and what knowledge would only be useful in rare situations.

I read Atlassian's Git guide, a Git for beginner's guide, and another guide from *Think Like (a) Git*. I watched a video called *Git for Ages 4 And Up*. While great resources that were newbie-friendly, these were still filled with concepts just not relevant to a technical writer.

After consuming everything Git-related and finally understanding it, I was able to figure out what information would be most useful to my team and develop training. Basically, the training ended up covering the common commands we'd use in our workflow and basic Git concepts like branching, merging, and tracking a repository's history instead of a file's changes.

Tools



Moving to Git affected the tools we use. We author our content in MadCap Flare 11, which offers integrated support for Git. We used this feature at first, but quickly decided to not bind our Git projects to source control in Flare. We encountered many issues with committing and merging changes, so we decided to separate source control from our authoring tool.

To track our Flare content with Git, we simply added each Flare project to a Git repository. We added some Flare project components to our .gitignore file, such as the Output and Analyzer folders and files Flare generates. We then authored our content within Flare and used other tools to handle source control.

We use both GitHub Desktop, a graphical user interface (GUI), and Git Shell, a command-line interface (CLI). An all-encompassing GUI client doesn't exist for Git, meaning you can't completely avoid the command line. GUI clients for Windows machines offer fewer commands in the interface, as we've sometimes painfully experienced with GitHub Desktop. However, we can accomplish most tasks in our workflow within the GUI. Though once you learn the commands, the command line is a very powerful and efficient tool that technical writers shouldn't fear.

I'm glad my team has started using Git. The ease of branching has sparked exploration and innovation. We're able to explore new writing styles, topic structures, and designs without worrying what to do if an exploration or wild idea takes a wrong turn.

Although Git has a steep learning curve, the best way to learn is to dive in and give it a try. Because Git tracks history, not individual changes, it's really hard to make disastrous, incurable mistakes. You can almost always go back to a better place in history and start again. Who wouldn't love that? i

MORGAN HANCOCK (morgan.hancock@bazaarvoice.com) *is a technical writer at Bazaarvoice in Austin, TX. She graduated from the University of North Texas in 2013 with a Master's degree in professional and technical communication. You can check out some of her technical explorations on Codepen* (http://codepen.io/mhancock) *and GitHub* (https://github.com/morganhancock).

Generating API Docs Automatically FROM THE Source Code

By ED MARSHALL | *STC Fellow*

shutterstock.com/ vasabii

THERE IS A COMMON PROBLEM in API (application programming interfaces) documentation that will occur in any group, no matter how well-meaning or conscientious the development and writing teams are. Sooner or later, the code will diverge from the documentation. Many developers favor an approach, coined using the "single source of truth" (SSOT). This means generating the documentation directly from the source code using specially formatted comments in the code. Typically, the primary use of this approach is to generate your API reference documentation. However, you can extend the tools used to extract the comments and generate your reference documentation and conceptual, "getting started" information, and flowcharts.

This article will introduce three tools commonly used to generate reference documentation from the code:

- Doxygen—Used primarily for C APIs
- Javadoc—Used primarily for Java APIs
- Swagger—Used primarily for REST APIs

Why should technical writers care about these tools?

API documentation is a rapidly growing area of technical communication. But it requires special skills, especially being adept with the same tools that developers use. But this field offers many opportunities for career growth and higher pay and prestige in your organization than other, more general areas of technical writing. Additionally, managers are often more amenable to "working at home"/remote telecommuting when you document APIs. Let's look at a few API basics first to give readers the context and then look at these tools.

What is good API documentation?

The standard baseline for good API documentation is a complete, accurate set of reference information that documents all the components of an API, especially methods. This dovetails very well with the approach of documenting an API in the source code.

Although you don't need to be a professional programmer or coder to be successful documenting APIs, reading the code is a key skill for documenting APIs, a major difference from documenting GUIs. You need to be able to read code well enough to recognize the following elements of an API in the source code: methods, parameters and datatypes, return values, and error codes. By reading the source code, you will find all the components you need to document. Additionally, Doxygen and Javadoc will verify that all components are documented when you generate your documentation output from those tools and report which components are not documented. Let's look at how three of the commonly-used tools to generate API documentation from the source code work.

Doxygen

Doxygen is a very powerful code generation tool that extracts specially formatted comments in the code and produces documentation. It supports C, C++, C#, Java, Python, and IDL. It outputs RTF, compiled HTML help, browser-based help, and LaTex (PDF). You can download it from *www.stack.nl/~dimitri/doxygen/download.html#latestsrc*.

Doxygen provides a set of predefined "tags," analogous to HTML tags, that identify the components and subcomponents implemented in an API. These tags are preceded by a special character, the backslash (\). The most commonly used tags specify parameters, return values, code snippets, notes, and cross-references to related components, usually methods. You can also use Doxygen and HTML tags to include conceptual, "getting started" information, and flowcharts directly in the source code files, and that content will appear in your generated output.

The following example shows the actual source code for a typical C method:

```
/**
  Opens a help topic in a specified help window.
  If a window type is not specified, a default
window type is used. If the window type or
default window type is open, the help topic
replaces the current topic in the window.

  \param pszFile Specifies a compiled help
(.chm) file, or a specific topic within a
compiled help file. To specify a defined window
type, insert a greater-than (>) character
followed by the name of the window type.

  \param dwData Specifies NULL or a pointer
to a topic within a compiled help file.

  \return The handle (hwnd) of the help window.

  \code
  HWND hwnd =
      HtmlHelp(
              GetDesktopWindow(),
              "c:\\help.chm::/intro.htm>mainwin",
              HH _ DISPLAY _ TOPIC,
              NULL) ;
  \endcode

  \note
  <UL>
    <LI> For backward compatibility with
WinHelp(), HH _ DISPLAY _ TOPIC and HH _ HELP _
FINDER provide the same functionality.
    <LI>A default help window contains only
the Topic pane and is not a three-pane Help
Viewer.
    </UL>

  \see HH _ HELP _ CONTEXT
*/
HH _ DISPLAY _ TOPIC(LPCWSTR pszFile, DWORD
dwData);
```

Note that all of your documentation comments precede the method in the code and appear in a standard comment block, "/* … */", except this comment block starts with a second asterisk, "/**". The second asterisk instructs the Doxygen compiler to compile everything in the comment block and include it in Doxygen's output. This example uses the following Doxygen tags:

- \param—Parameters are the variables you specify when calling a method. You need a param tag for each parameter defined for your method.
- \return—If your method returns a value, you must document it using this tag.
- \code … \endcode—This is your code snippet. Doxygen preformats the example and puts it in a monospaced font, commonly used for examples in API documentation.
- \note—This is nice-to-know, supplemental information, not needed all the time, but occasionally.
- \see—These are your cross-references to related methods. If you have multiple cross-references, precede each one with a \see tag.

Note that there is a bulleted list coded in HTML in the code example. Doxygen supports most of HTML tags so you can use those tags for formatting, as well as the tags defined in Doxygen.

Figure 1 shows the Doxygen output for the HH_DISPLAY_TOPIC method. Note the output format of this example. Doxygen provides its own cascading style sheet (CSS) and predefines the output layout. The actual method with its syntax appears first, followed by a brief description of the method. The method's parameters appear in the order they appear in the syntax, the method's return value, followed by a code example showing the use of the method, some notes or supplemental information, and lastly a cross-reference to a related method. This is the standard outline or organization for API reference information, regardless of company or tool used.

Javadoc

Javadoc is a powerful and free code generation tool for Java APIs. As with Doxygen, it reads specially formatted comments in code. It outputs what many of the popular help authoring tools (HATs) call browser-based help, with some features that our common tools do not provide. These will be pointed out in the following examples. It also is actively supported. You need to download and install the Oracle JDK (Java Development Kit). The current version is 8.0, but the author recommends you use JDK 7 as 8.x requires ending all tags with end tags. If you are working on legacy documentation, which is likely, you will have to clean up your Javadoc comments, which could be a daunting effort in JDK 8.x without generating large numbers of warnings. You can download the JDK from *www.oracle.com/technetwork/java/javaee/downloads/index.html?ssSourceSiteId=ocomen*.

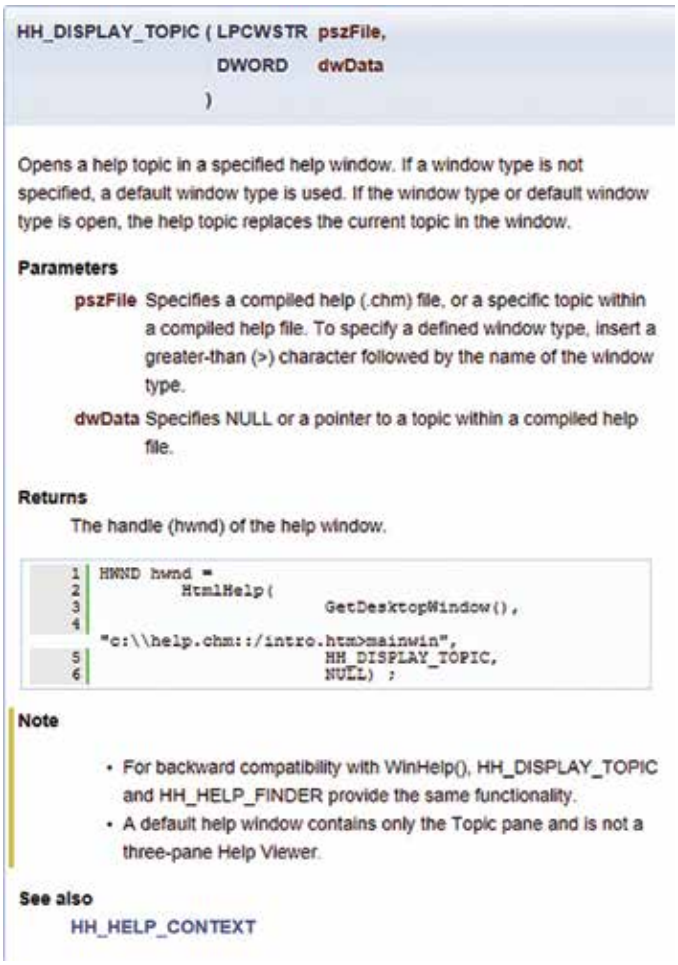

HH_DISPLAY_TOPIC (LPCWSTR pszFile,
DWORD dwData
)
Opens a help topic in a specified help window. If a window type is not specified, a default window type is used. If the window type or default window type is open, the help topic replaces the current topic in the window.
Parameters
pszFile Specifies a compiled help (.chm) file, or a specific topic within a compiled help file. To specify a defined window type, insert a greater-than (>) character followed by the name of the window type.
dwData Specifies NULL or a pointer to a topic within a compiled help file.
Returns
The handle (hwnd) of the help window.
HWND hwnd =
HtmlHelp(
GetDesktopWindow(),
"c:\\help.chm::/intro.htm>mainwin",
HH_DISPLAY_TOPIC,
NULL) ;
Note
For backward compatibility with WinHelp(), HH_DISPLAY_TOPIC and HH_HELP_FINDER provide the same functionality.
A default help window contains only the Topic pane and is not a three-pane Help Viewer.
See also
HH_HELP_CONTEXT

Figure 1. Doxygen output for HH_DISPLAY_TOPIC method.

Javadoc provides a set of predefined "tags" that specify the components and subcomponents implemented in an API. These tags are preceded by a special character, the at symbol (@). The most commonly used tags specify parameters, return values, code snippets, notes, and cross-references to related components, usually methods.

Figure 2 shows a sample method, login, for a Java API. As with Doxygen, all of your Javadoc documentation comments precede the method in the code. They appear in a standard comment block, "/* … */", except this comment block starts with a second asterisk, "/**". The second asterisk instructs the Javadoc compiler to compile everything in the comment block and include it in Javadoc's output. This example uses the following Javadoc tags:

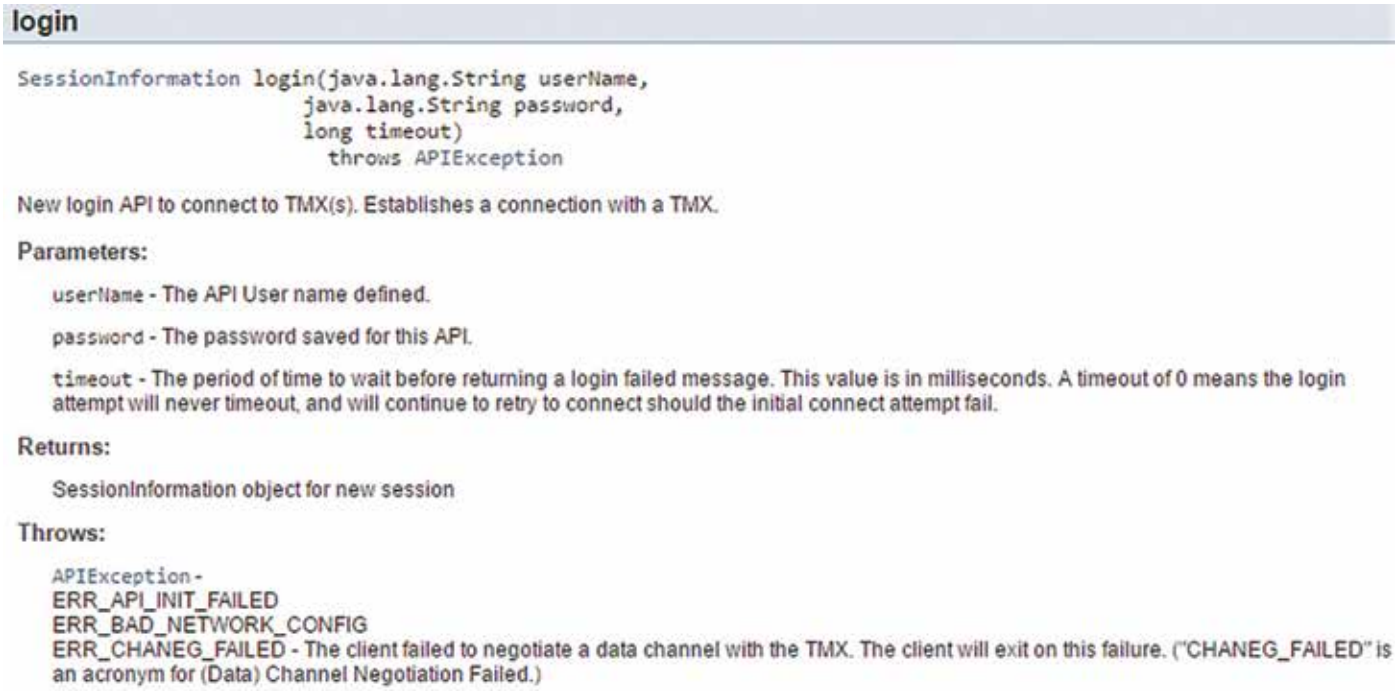
- @param—Parameters are the variables you specify when calling a method. You need a param tag for each parameter defined for your method.
- @return—If your method returns a value, you must document it using this tag.
- @throws—In Java, when an error is thrown, it is called throwing an exception, so Javadoc uses a throws tag.

Javadoc supports most HTML tags so you can use those tags for formatting, as well as the tags defined in Javadoc.

```
/**
    * New login API to connect to TMX(s).
    *
    * Establishes a connection with a TMX.
    * @param userName The API User name defined.
    * @param password The password saved for this API.
    * @param timeout The period of time to wait before returning a login failed message.
    * This value is in milliseconds.
    * A timeout of 0 means the login attempt will never timeout,
    * and will continue to retry to connect should the initial connect attempt fail.
    * @return SessionInformation object for new session
    * @throws APIException
    * <br>ERR_API_INIT_FAILED
    * <br>ERR_BAD_NETWORK_CONFIG
    * <br>ERR_CHANEG_FAILED - The client failed to negotiate a data channel with the TMX.
    * The client will exit on this failure.
    * ("CHANEG_FAILED" is an acronym for (Data) Channel Negotiation Failed.)
    */
   SessionInformation login(
       String userName,
       String password,
       long timeout,
       throws
APIException;
```

Figure 2. A sample method (login) for a Java API.

Figure 3 shows the Javadoc output for the login method. Note the output format of this example. Javadoc provides its own cascading style sheet (CSS) and predefines the output layout. The actual method with syntax appears first, followed by a brief description of the method. The method's parameters appear in the order they appear



login

```
SessionInformation login(java.lang.String userName,
                         java.lang.String password,
                         long timeout)
                           throws APIException
```

New login API to connect to TMX(s). Establishes a connection with a TMX.

Parameters:

`userName` - The API User name defined.

`password` - The password saved for this API.

`timeout` - The period of time to wait before returning a login failed message. This value is in milliseconds. A timeout of 0 means the login attempt will never timeout, and will continue to retry to connect should the initial connect attempt fail.

Returns:

SessionInformation object for new session

Throws:

`APIException` -
ERR_API_INIT_FAILED
ERR_BAD_NETWORK_CONFIG
ERR_CHANEG_FAILED - The client failed to negotiate a data channel with the TMX. The client will exit on this failure. ("CHANEG_FAILED" is an acronym for (Data) Channel Negotiation Failed.)

Figure 3. The Javadoc output for the login method.

By learning how to use tools such as Doxygen, Javadoc, and Swagger that extract specially formatted comments from the code, we enhance our value to our organizations and to our users.

in the syntax, the method's return value, and the errors returned for this method.

Figure 4 shows the main topics in a Javadoc system. The Overview is the "welcome page" to your Javadocs, by default. You can also use Javadoc and HTML tags to include conceptual, "getting started" information, and flowcharts directly in the source code files and that content will appear in the Overview topic.

Overview Package Class Use Tree Deprecated Index Help
Prev Letter Next Letter Frames No Frames
A B C D E F G H I L M N O P Q R S T U V W

Figure 4. The main topics in a Javadoc system.

Another useful and unique feature of Javadoc is that it automatically creates an index of all your API components where users can search for an API component by name. Each entry in the index is hyperlinked to the documentation for that component. If you click on the letter L, Javadoc will display all the methods that start with the letter L. So, once you find the component in the index, you can click on the link to go directly to the information on that component. This is a very useful feature for developers as they can quickly find a method if you know its name or see all the methods starting with a chosen letter.

Web Services APIs

Web services APIs are a newer type of APIs but are becoming very popular. There are two types:

1. Representational State Transfer (REST)—All parameters and values called are specified in the request. To call a REST API, you specify a base URI (URL): *http://example.com/resources/.*

   REST APIs use standard HTTP actions for their operations:
   - GET—Lists/retrieves data from a record. Doesn't change the record.
   - POST—Posts/stores or writes data into a record. Creates a new entry in a collection/data in a record.
   - PUT—Replaces the entire collection with another collection.
   - DELETE—Removes data from a record. Performs the same operation every time.

2. Simple Object Access Protocol (SOAP)—A method is predefined with all request parameters and values specified. You do not specify which of the four HTTP actions to call for a method. Each method is predefined to use a specific action. For example, the getCustomerInfo method uses the GET HTTP action.

Swagger

Swagger is a popular tool for documenting Web service, APIs. It not only extracts documentation from your source code but has the added benefit of providing an interactive way of trying the API. That is, you can specify values for the parameters and "test-drive" the API by calling the resource and see the result.

An excellent interactive Swagger example can be found at *http://petstore.swagger.io/.* This example provides a pet store application where you can add a new pet to the store, update an existing pet, find pets by status, tags, or ID, delete a pet, and perform other operations. The Web-based UI is populated from the source code.

Let's look at an extract from that example. The following shows the code for adding a pet:

```
"schemes":["http"],"paths":{"/
pet":{"post":{"tags":["pet"],
"summary":"Add a new pet to the store",
"description":"","operationId":"addPet",
"consumes":["application/json","applica-
tion/xml"],
"produces":["application/xml","application/
json"],
"parameters":[{"in":"body","name":"body","-
description":"Pet object that needs to be
added to the store",
"required":true,
"schema":{"$ref":"#/definitions/Pet"}}],
"responses":{"405":{"description":"Invalid
input"}},
"security":[{"petstore _ auth":["write:-
pets","read:pets"]}]},
```

Figure 5 shows the Swagger UI for adding a pet. The body field on the left is where you specify values to test; the Model Example Value field on the right shows the syntax. In this case, I added a new pet named Misty with an id of 10.

To test the interactive aspect of Swagger, click the "Try it out!" button in the lower left to display the response and any response codes to your request.

The Response Body section shows that a pet named, Misty, with an id of 10 was added. The Response Code indicates if the operation succeeded or what that problem was. In this case, the response code is 200, indicating success (see Figure 6).

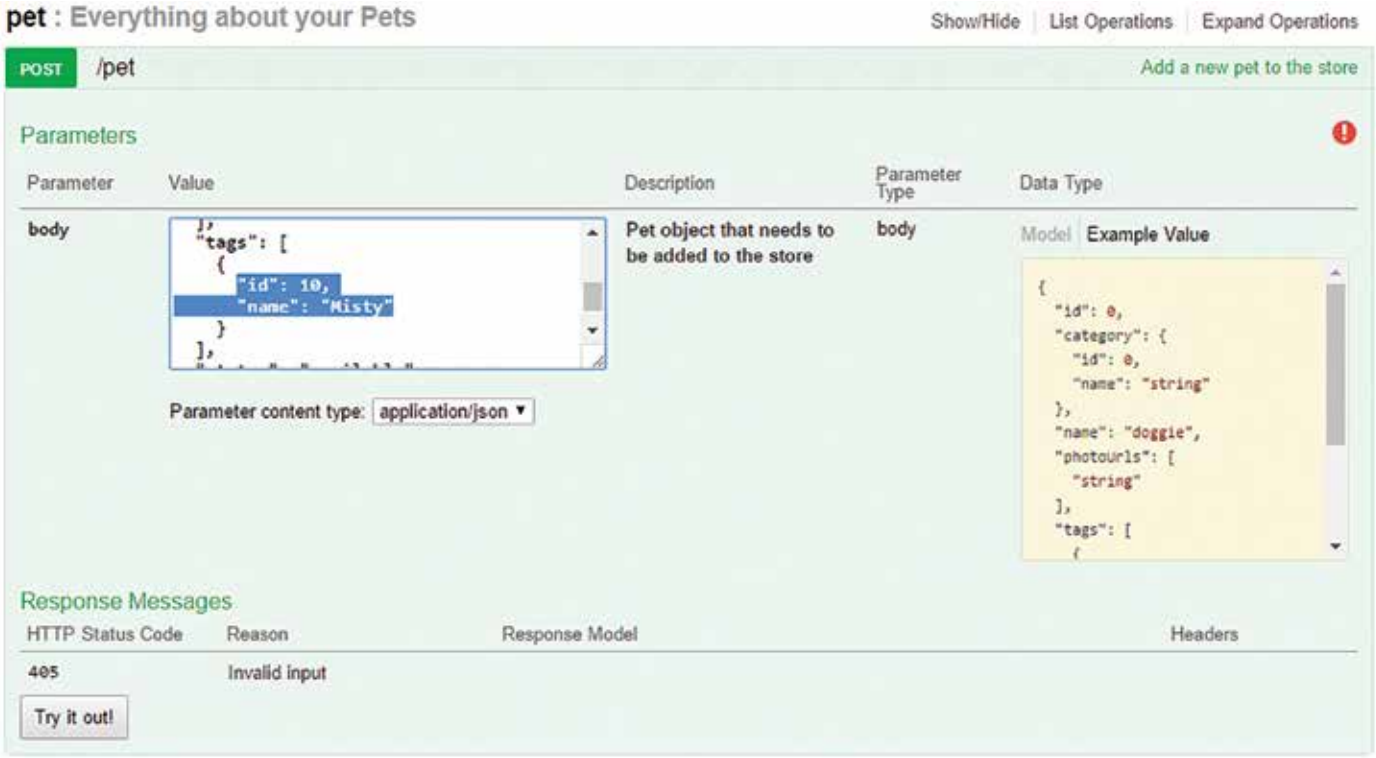


Figure 5. The Swagger UI for adding a pet.

Figure 6. The Response Body and Response Code.

Summary

Developers prefer to generate the reference information for an API directly from the source code to help ensure that all the components of an API are documented and nothing was missed. Otherwise, the code and documentation will get out of sync eventually. Someone, either a developer or a technical writer, has to add those comments. We, as technical writers, can provide extra value by using the writing, analyzing, editing, proofreading, and organizational skills we've learned and use in other areas of technical writing. By learning how to use tools such as Doxygen, Javadoc, and Swagger, that extract specially formatted comments from the code, we enhance our value to our organizations and to our users.

ED MARSHALL *(*ed.marshall@verizon.net*) is an independent consultant technical writer and the sole proprietor of Marshall Documentation Consulting (*www.MarshallDocumentationServices.com*), with over 28 years of experience. He specializes in technical documentation for developers, including APIs (application programming interfaces), SDKs (software developer's kits), Web services products, etc. Over his career, he has developed expertise in using tools to "let the computer do the work," such as advanced tools for editing files, comparing files, and searching and replacing text. He is an STC Fellow and has given many presentations at local STC chapters and Summits, the WritersUA conferences, Information Development World, tekom, and many other events. He is a Certified MadCap Advanced Developer (MAD). He can be reached on LinkedIn at* www.linkedin.com/pub/ed-marshall/0/501/898 *and on Twitter* @EdMarshall.

Creating Automation Tools for Writing Teams: A Programmer's Perspective

shutterstock.com/ Aha-Soft

By ROBERT DELWOOD | *STC Senior Member*

TOOLS WILL ALWAYS be the weak link for writers. Not the high-level ones such as Flare or Framemaker, but low-level tools, ones that writers need to finish individual tasks. These tasks may be unique to each group or perhaps to each writer, from reformatting long lists of error codes, to collecting acronyms within a document, to special or conditional formatting on tables, or repeating onerous sequences—a representative list is impossible to create. Given the extreme diversity among writer's procedures, it is unlikely we will get a comprehensive tool suite. Neither the commercial sector nor the writing community at large provides these suites. That means we have to write them ourselves.

We've always been told that the goal is never far off. Many high-level tools have at least some support for this. Notably, Microsoft Word has a built in macro language (VBA), a macro recorder, a code editor, and complete programmatic access to its commands. These tools, though, are still very much in a programmer's realm, out of reach for many writers, and development groups rarely loan out programmers. But what if they did? What results are possible?

I was fortunate to be assigned to three writing groups at the NASA Johnson Space Center in Houston for a number of years. The mission was simple: to improve these teams' efficiency. I had considerable leeway in my methods, but mostly through the creation of custom tools and procedural changes. My qualifications seemed fitting. I was a programmer, writer, and programmer-writer, and thus familiar with writers and the writing process. The teams were grateful for the attention and willing to help although there were several conditions that had to be addressed.

Initial mistrust. There was serious mistrust about automation, with writers imagining that it would eliminate their jobs. The truth is, it's about freeing team members to do what they're good at. Start on a project they are comfortable with and earn their trust. Let them set requirements, provide frequent product updates, show progress, and offer options to alleviate those initial fears.

User education. Most team members are Word power users and capable of solving Word problems themselves. Automation doesn't minimize their skills. Quite the opposite; they still have to tell you to construct the documents and requirements. Over time, they'll learn what is possible with automation. The best ideas come from the team members themselves, but they have to know what can be automated and they have to be comfortable enough to ask. My favorite part of any project is when writers start asking, "Is this possible?" or "Can we do this?"

Bureaucracy. NASA is a government agency and despite the perception of being innovative, it's ultimately still a bureaucracy. Combined with other factors such as the reliance on international partners, interdepartmental coordination, and natural resistance to change, it was not always possible to make the right change. We had to work within what we could change. Don't be afraid to compromise or, conversely, overstep boundaries a little as needed. Remember, it's easier to keep a project that exists, rather than sell just an idea for a new one.

Development. Typical writer's automation projects tend not to be large or take a long time, perhaps two or three weeks. They have very specific goals, often just a single task. One project took less than two hours. In a way, it may not be much more than rapid prototyping. Working directly with the clients provides immediate feedback and shortens development time.

Once completed, you deploy projects in one of two ways. If the project is simple enough, it can be written as a series of macros and deployed through a common template. Otherwise, it will be deployed as a Windows application. This adds an installation step, but the application benefits from .NET features, making it more versatile. Web deployment was not an option for us because Microsoft Office then couldn't be used over an intranet.

What Can Be Automated?

I advocate that almost every team can use a programmer-writer. The team may not even know it, but after pointing out the opportunities, tools become indispensable. Think about your procedures and which ones are repeated, either immediately or periodically, such as before releases. Repetitious steps are obvious candidates for automation. Some are less obvious. For example, Flare truncates bookmark names to eight letters. As a result, sections such as "Installing System Drivers" and "Installing System Tools" display as "Installi" and "Installi2." This truncation is annoying and unhelpful. You can manually change these one by one, but an automated tool would fix all the occurrences in only a few seconds. You could also, for example, use a building block or snippet to create a

preformatted table, but over time the formatting might change. Instead, you could write a macro ensuring the format, from the table headers to the font and size for each cell, is correct. If you run that macro on dozens of tables before each release, the time and quality savings become significant.

Experienced programmer-writers recognize these opportunities. They can help point out what is and isn't automatable, and they often know procedural changes to accompany the automation.

The following are three representative but diverse projects.

TIFF-based Application

This is an example of controlling two applications, merging data from one to another. The first document is an enumerated list of parts and equipment going up to the space station. The restrictions were to create a computer-generated RFT file that couldn't be handled like a normal Word document; it had to be read only, and each page had to paste individually into a master, or target document, complete with table titles and body text provided by engineers. The conventional way of producing this was to save the list document as a single TIFF image file. Each page was then hand copied from Photoshop, pasted, and resized into the target. The list document was often more than 80 pages, and the handbook rated this procedure at 45 hours. In other words, it took one person more than a week of repetitious, error prone processing per iteration, and they may have had five to eight documents a year. In addition, a quality assurance (QA) team member checked each page, ensuring completeness and no duplicates.

This was an ideal automation project. The steps were well defined, there were no exceptions to the business logic, and it had a repetitive nature. Deployed as a Windows application, the team member started by selecting the list document, and placing the cursor in the target document where the first image was to be placed. The application would then image the list document. Because it was an RTF file, page breaks determined individual images. It generated each page as a separate TIFF file. When the images were complete, each file was opened automatically, and the image was copied and pasted at the insertion point in the target. The application also automatically resized each image for the available page, determining page dimensions and allowing for titles. The insertion point automatically advanced to the next page and the process repeated. The document completed in less than 15 minutes. After validating the process for a few weeks, they dropped the quality review requirement, saving additional time.

Acronyms

The most onerous requirement was that each document needed an acronym list. This list included identifying acronyms, cataloging them (complete with the definition), spelling them out on first occurrence, and providing a revision table to show newly added, deleted, or changed terms. Desktop guides rated this task as 32 hours per document, although it often took longer, sometimes more than 40 hours. Over time, the number of flights to the space station increased from 3 to over 15 a year, and bureaucracies being what they are, this meant that the documentation increased at a disproportionately high rate. Therefore, it was unreasonable to expect this process to remain a manual effort.

The difficulty started by just identifying what an acronym was. It could actually be anything from a three-letter acronym (USA or ARS [Air Revitalization System]), to terms (Log [Logistics], ATT [Attitude]), or almost anything else (rack loc [rack location], Rqmt [Requirement], or *U.S. Lab*). The approach had been simply looking at each page, usually printed, and using the reviewer's skill to spot acronyms. It was easy to assume the Find dialog could have been used, but reviewers still had to copy and paste each term, and the list approached 2,500 terms, double that if you included the definition. And that only covered the known terms. Engineers could introduce new terms without notification. There was also spelling out the term on the first occurrence, which became increasingly frustrating since material was often moved. The definitions had slight variations that had to be corrected (Heat Rejection Subsystem/Heat Rejection System, or Subelement/Sub-Element). As a result, automation had to be used, not only for quality control but, more critically, to keep up with the increasing workload.

The automation occurred in three steps because each step required verification and validation. The first step found and listed all the acronyms. Ultimately, we defined three ways to find them. What was called the "master list" was a complete term listing of 2,500 terms. It continues to grow and is controlled by select book managers. The application searches for these terms explicitly first. Then it searches the existing document's acronym list, assuming that those terms were already correct. Finally, we created a catchall using Word's internal misspelling dictionary. This identified all other terms, assuming an acronym shows up as a misspelling. For that list, we reviewed each term, adding the acronyms as needed. We managed those terms using Word's dictionary and exclusion dictionary (such as for people's names like *Sufferdini*). Some acronyms were also legitimate words (like *temp* for temporary, or *He* for Helium). Unless those terms were in the master list or the document's current acronym list, they had to be found manually. Regardless, we achieved nearly 100% accuracy.

The second step produced the acronym list as Word table, complete with revision tracks and style formatting applied. Upon approval, it was this table that was pasted into the target document.

The third step created a first occurrences report that team members used to verify that the term was spelled correctly and fully on first use.

What Is VBA?

VBA is the programming language Visual Basic for Applications. It is form of Basic and is widely held to be an easy language. It reads like English and can be programmed with less difficulty than other languages, like C or C#. VBA is not a modern language like Microsoft .NET, being a predecessor to VB.NET, and is limited in some advanced functions.

However, VBA has the advantages of still being a versatile language and, most notably, it's built into all Microsoft Office applications, along with a code editor. This implies two things. First, it supports Microsoft's OLE Automation, which means it interacts with all Office applications. Second, it is an interpreted language rather than compiled. VBA has to be hosted inside an application, typically Office applications like Word, and can't be used as a standalone application or as an *.exe file.

You don't need to buy a compiler or editor for it, because it's fully incorporated into Office applications automatically. The macro tool—the one that lets you record macros as a sequence of commands—is also a VBA code generator. For example, after recording a macro, the code will be VBA. You can then edit that code as original programming.

Creating a Macro

An important automation aspect of Word is its ability to create and run macros. Macros are the ability to record a sequence of commands, and then play them back in the same sequence. This has important time saving and quality implications in that you can record a detailed and complicated sequence and play them back with a single keystroke. To get started:

1. Open Microsoft Word.
2. Click **View | Macros | Record Macro**. The **Record Macro** dialog displays.
3. You may choose to rename it from *Macro1*.
4. Select **Document1** from **Store macro in** drop down.
5. Click **OK**. The mouse icon changes to include a cassette, indicating you're recording.
6. Perform your commands. In this example, we'll format a table.
   a. Type "Macro Example." Highlight it and style it as Heading 1.
   b. Place the cursor at the end of the line and enter Return.
   c. Select **Insert | Table | Insert Table**, sweeping a 2x2 table.
   d. Enter text in each of the four cells. You will need to tab to the next cell rather than using the mouse.
   e. Select both cells of the first row. You'll have to use the keyboard arrow keys to move the cursor.
   f. Select any of a different font, size, color, or style.
7. Click **View | Macros | Stop Recording**.
8. Delete everything in the document.
9. Click **View | Macros | View Macros**.
10. Double click the macro name. The macro runs and the table is placed exactly as you entered it.

The new automation took between 20 minutes to an hour, depending on the complexity of the document, with a turnaround time reduced to less than four hours overall. Not all the changes were through software. One important process modification was to move this entire procedure from the quality review team to the book managers, as they could fix term anomalies quickly. This level of accuracy and speed is not possible manually. It had to be an automated tool. Neither could have been written as a series of macros. For instance, because of the sheer number of internal data iterations and algorithms, it had to be optimized for speed. It also required accessing Word's internal structures.

The Importance of Being Embedded

The traditional programming approach keeps the programmer separate from the client. However, in programming for writers, it is critical to embed programmers with the team. Merely sitting with them has advantages, such as overhearing conversations, learning jargon, observing tasks, and noting how things are being done. For example, in one instance I noticed a team member repeatedly hitting the same key for more than a minute. She was transferring files and her application didn't accept long names, so she had to skip those. Furthermore, there were at least 5,000 files in the archive, so that task would have taken an excessive amount of time, including manually changing the names to fix them. By intervening, we created an ad hoc application in less than thirty minutes that checked all the names directly in the Windows depository and corrected the long ones. A day-long task took less than two hours (with testing) and could be used by other team members. Even though she was an experienced team member who had worked with me before, she didn't realize that the task could be automated at the Windows level.

An extreme example happened later when I was transferred to become part of a team—a meeting support team that organizes routine formal management and engineering meetings, provides transcriptions and voice recordings, and manages agendas and invitation lists. Due to their transparency, they had been overlooked for technology and process improvements. The mission was essentially process improvement, changing procedures as needed, and writing efficiency tools. "Nothing is off the table," said my supervisor. This was truly an exciting and evolutionary step. In the nine-month assignment, the first three months were to learn to do the job. After that, changes could be made. And the insight that period provided was revolutionary. In all, we added 14 tools. They ranged from converting email forms to meeting logs, automatically adding and modifying agenda items, and queuing long lists of presentations (from PowerPoint, Word, and Excel) into a single PDF and print job. Procedures were changed, too, such as pooling team members to help with other meetings based on their availability, to something as simple as combining Windows locations into a single directory.

Failed Projects: Table Number 3

Not all projects, even those considered good automation ones, succeed. Table Number 3 in one of our documents was a notable failure. It was a standard table, in this case, vehicle launch, docking, and return information for all the flights (manned and supply missions) to the space station during a given period. The problems started when we discovered the business logic (that is, the rules applied to formatting and notes presentation) changed after each release. This was partly due, ironically, to our success, because managers wanted more business logic. But it was largely because the rules changed outright, as flight engineers redefined them. In addition to writing new rules, the existing business logic became almost incomprehensively complex, including exceptions with exceptions to the exceptions. Due to the time and effort required to maintain it, management decided to revert the automation to manual formatting. The code was rolled back to an earlier phase of creating and formatting the table for its initial use.

Summary

Automation is an extremely powerful tool and can be applied in innumerable ways. For one writing team, the document turnaround time dropped from the contractually required twenty days to just three. In addition to saving time and money, these tools reduce stress. Tools development doesn't have to be complex. Word encourages creating macros just by recording your steps. However, adding a programmer can unlock additional features, such all of .NET, regular expressions, and internal Word capabilities. It is unlikely that you would be assigned your own programmer, though, so consider approaching the IT department with a specific request for a single feature. As a compromise, suggest building complex tasks progressively. A small investment from a programmer could save the writing team literally weeks each year.

ROBERT DELWOOD *is a programmer, writer, and programmer-writer formerly with NASA's Johnson Space Center and Microsoft. With more than 18 years' experience, he has written and documented topics from Windows kernel-level device drivers and speech recognition APIs/SDKs for Microsoft, to help desk procedures and application manuals for the military. He specializes in Microsoft Office automation with VB/VBA and .NET VSTO. He's authored several books, the most recent one, a college-level textbook,* The Secret Life of Word *(*http://xmlpress.net/publications/word-secrets*/) about Word's automation for technical writers, non-programmers, knowledge workers, or anyone who wants to do more tasks quickly with Word.*

"I MET MANY OF MY FAVORITE PEOPLE THANKS TO STC"
"When I first joined STC and started going to meetings, I quickly realized that STC provided some of the best networking and professional development opportunities for our profession. I was coming from working in a veterinary clinic, which is a totally different world, and I didn't have a lot of contacts in technical communication. Today, most of my work friends and some of my closest personal friends are a direct result of being an active volunteer in STC."
Kit Brown-Hoekstra
MY NAME IS KIT BROWN-HOEKSTRA AND
I'M AN STC MEMBER
STC.org/renew

Using Data-Driven Synthesis Tools to Automatically Generate Content

shutterstock.com/ file404

By JOEL KLINE | *STC Senior Member* and FRANK GUERINO

Introduction to Data-Driven Synthesis (DDS)

Most technical communication (TC) practitioners are familiar with content authoring tools that generate, curate, and publish content one page at a time, such as content management systems (CMSs) and wikis. Most of us are also familiar with software compilers that convert software source code into executable software that can be installed and run on computers (e.g., Java and C++ compilers). This article is about a rapidly evolving paradigm called Data-Driven Synthesis (DDS). Like software compilers, DDS uses tools we call *data compilers* to ingest data and automatically generate very large volumes of content which could not otherwise be written by human hands without significant time, effort, and funds.

Consider, for example, being assigned 1,000 products that you have to fully document across your company's website, intranet, and product literature. Imagine requiring at least one page per product. Next, imagine the work of creating all 1,000+ pages, formatting them, creating catalog pages, creating indices, and maintaining all the links for all these product pages. Even if you use a database-driven solution, you still encounter challenges when the data structure changes, such as when each product gets a Twitter handle.

DDS is a paradigm that allows TC practitioners to author content faster, with much higher levels of quality, and for a fraction of the costs.

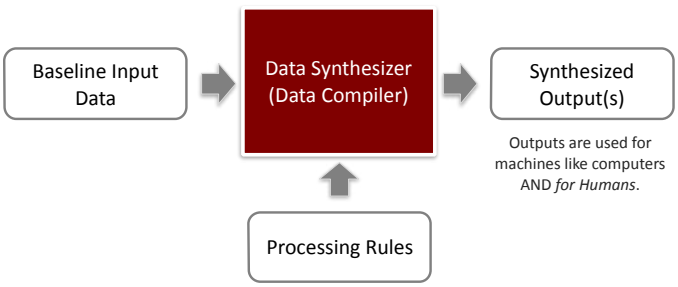

Baseline Input Data
Data Synthesizer (Data Compiler)
Synthesized Output(s)
Outputs are used for machines like computers AND *for Humans*.
Processing Rules

Figure 1. Basic Data-Driven Synthesis (DDS) paradigm.

Synthesis is not new

Synthesis, as a general concept, is not new. Compilation of computer source code, for the purpose of generating executable computing constructs (i.e., software), dates back to the 1950s. Synthesis of semiconductor simulation, emulation, and fabrication constructs from 4GL languages such as VHDL and Verilog has been the norm since the mid-1990s.

More recent examples of synthesis can be found in solutions such as software scaffolding frameworks like Ruby on Rails, in big data analytics, in visualization generation solutions like Data-Driven Documents (D3.js), and in predictive analytics solutions that help automatically calculate things like healthcare pathways for patient treatments and pharmaceutical drugs for targeted clinical outcomes.

DDS works for both tiny data and big data

Data scientists commonly use DDS to automatically perform complex informatics work that involves looking for patterns and anomalies in very large sets of big data (often many petabytes). Many TC practitioners often deal with much smaller data sets that are usually no bigger than gigabytes, often referred to as tiny data. In our example of managing 1,000 products, the amount of data is small compared to big data, but the challenges lie in publishing and updating.

The Wikipedia Problem

Technical communicators have used wikis for content generation since the advent of Web 2.0. While the value of wikis for collaborative writing is undisputed, their use for content generation within the enterprise produces numerous challenges. Enterprises often cite Wikipedia as *the* example for how to decentralize content generation and provide everyone with a role in creating content. However, this decentralized model of content generation rarely—even with oversight from technical communicators—yields the volume and quality of content necessary for success.

Let's briefly examine why a Wikipedia paradigm for enterprise content generation is problematic. The primary problem with Wikipedia for enterprise authoring is its Long Form Notation (LFN) structure. This leads to many authoring problems. One such problem is scale. It takes many people long periods of time to write content for wikis. Most enterprises have a difficult, if not impossible, time getting employees to write wiki entries. A second problem with wikis is indexing. LFN makes it hard to index the content and difficult to create relationships between data and its attributes. A third problem with wiki content is repurposing. LFN makes Wiki content difficult to automatically repurpose because of the absence of data attributes. The final problem with wiki content is updating. Although the hyperlinked structure makes link updates easy, the LFN in a wiki makes it challenging to locate and update individual data elements.

The paradigm of Data-Driven Synthesis uses automated content generation to solve the problems presented by the Wikipedia paradigm. DDS methods use your data to create very large volumes of high quality content, quickly. In this article, we show how one dynamic compiler converts anything in a Comma Separated Value (CSV) format to HTML documentation to display different knowledge constructs and dependencies to your audience using Short Form Notation (SFN). SFN eliminates long verbose paragraphs that are associated with LFN by replacing them with sets of succinct name-value data pairs. With simple data management constructs like spreadsheets or other flat files, DDS data compilers use these name-value pairs to format and connect data. The compilation feature for DDS is one of the paradigm's major strengths. When data changes, just recompile and republish. This makes it easy to quickly change and repurpose large volumes of content.

If data-driven synthesis can solve your automatic content generation problems, what's holding you back? In the next section, we present a short argument for learning to employ dynamic data-driven synthesis for content generation in your enterprise.

DDS Bridges Technical Communication and Big Data

You cannot read a current periodical in the fields of business, information science, or communication without seeing the term big data. Big data and its discipline, data science, is an important concept for technical communicators who generate content. But not all topics related to data science are central to technical communication. In fact, tiny data is more relevant to TC professionals than big data. Most TC practitioners will not be accessing petabytes of data in order to generate content. Instead, they commonly deal in data sets that are hundreds to thousands of records per topic area (e.g., our recurring example of 1,000 products with 100 descriptive traits). This is far less complex, for example, than business intelligence activities that require analysis of millions of daily transactions. However, big data methods can be useful to TC professionals, even when working in tiny data spaces. Big data methods can facilitate the automatic generation of large quantities of higher quality content, such as richly formatted views of content and different knowledge constructs that include data visualizations. As noted earlier, DDS paradigms can overcome Wikipedia paradigm problems associated with LFN such as scaling, indexing, repurposing, and updating. An additional benefit to the DDS paradigm is the ability to synthesize data visualizations so that technical communicators do not have to write code to create a visualization.

DDS provides an entry point for technical communicators to harness the power of data visualization. When content is in a dynamic format, it is possible to output to numerous kinds of data visualization schemas. Data visualization occurs when the synthesizer (i.e., data compiler) discovers data attributes, data formats, and data relationships via rules specified by the TC practitioner generating the outputs. DDS-generated data is typically structured as Short Form Notation that contains a name-value data pair (i.e., a descriptive attribute and its data value). Consider our 1,000 products example. DDS permits the technical communicator to combine product data with other enterprise data to easily generate visualizations that tell stories about the products. For example, visualizations can be generated that show relationships between products and other important data entities, such as people, documents, organizations, videos, and images.

In a DDS system, a technical communicator can compile CSV flat files to produce visualizations that represent the many different things which are related to each product. For example, your enterprise may have different support documents, videos, links, images, and people associated with specific products. The data compiler generates content that helps visualize different traits for each product instance. In Figure 2, the data compiler generated visual relationships between a product and other things it is related to (e.g., applications, contracts, other products, human resources, etc.).

Incorporation of DDS paradigms is critical for making the transition from static documentation to dynamic documentation. This change requires that the technical communicator identify relationships between data and its attributes. However, once those relationships are identified, the compiler now has the capacity to generate all kinds of dynamic output, including vast quantities of richly formatted content, various knowledge constructs, and bundled package sets like digital libraries. Our next section describes the application of DDS Synthesis to these specific areas.

How to Use DDS for Technical Communication

We've found that applying DDS to automatically generate electronic documentation has direct and positive impacts on the productivity of TC professionals. Tools like the IF4IT

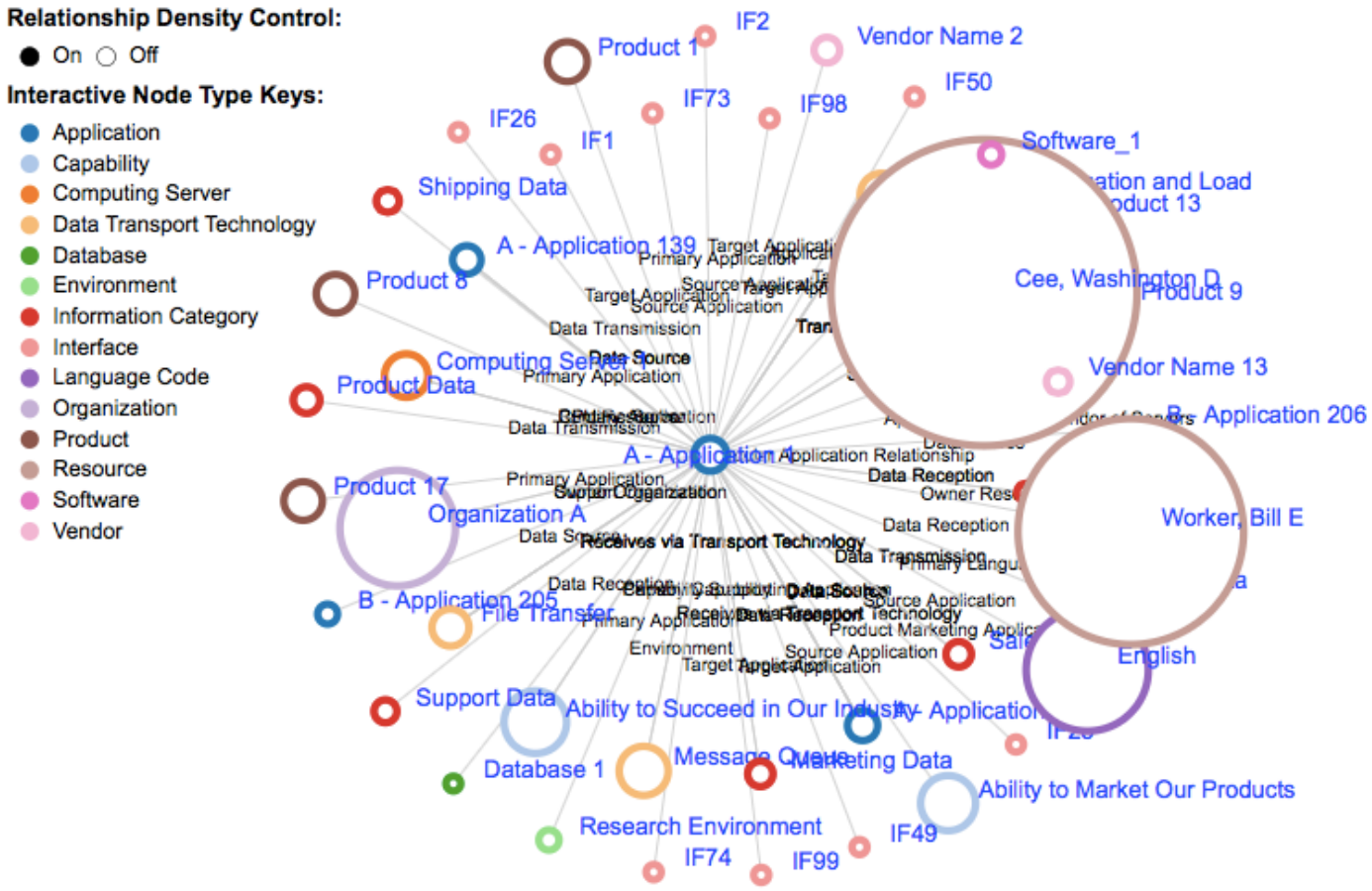


Figure 2. Example visualization for a product and its relationships.

NOUNZ Data Compiler use DDS to automatically generate interactive electronic documentation, thus empowering a single person to outperform large human labor forces. Data compilers ingest structured data and use processing rules to produce at least three key outputs that are useful for TC practitioners:

1. Large volumes of richly formatted content.
2. Knowledge constructs, including data visualizations.
3. Digital libraries.

Synthesis of large volumes of richly formatted content

Let's return to our example of authoring content in support of 1,000 products. Traditionally, there were only two options for building consistent and accurate Web pages for these products. The first option was to leverage tools that allow human labor forces to create and format each page, one at a time (e.g., content management systems). The second option was to build custom software solutions.

The first option requires lower levels of skill but is limited by the performance of humans. It is slow and prone to significant human error. The second option requires complex skills to design, deliver, and maintain the custom system. Both options are expensive and are tied to long delivery times.

DDS based solutions offer a third option. Data compilers like NOUNZ allow a TC professional to easily submit their data to the compiler and automatically generate electronic documentation views with little effort and with just a few minutes of work (see Figure 3).

In authoring content to support 1,000 products, the technical communicator would submit product data as CSV flat files to a data compiler that would automatically create more than just 1,000 individual product pages of content. In addition to creating descriptive content pages that mostly use text, the data compiler can also generate interactive data visualization views that highlight and compare data relationships between products and other important data types. A data compiler can easily generate many different views that humans would otherwise not have the time or funding to create manually or via custom application development. Data compilers also ensure that pages are consistently formatted so that the user experience makes it easier to explore, discover, and learn.

Synthesis of knowledge constructs (KCs) such as data visualizations

In addition to the generation of many content pages, DDS-based data compilers also generate both simple and complex knowledge constructs (KCs). KCs are interactive

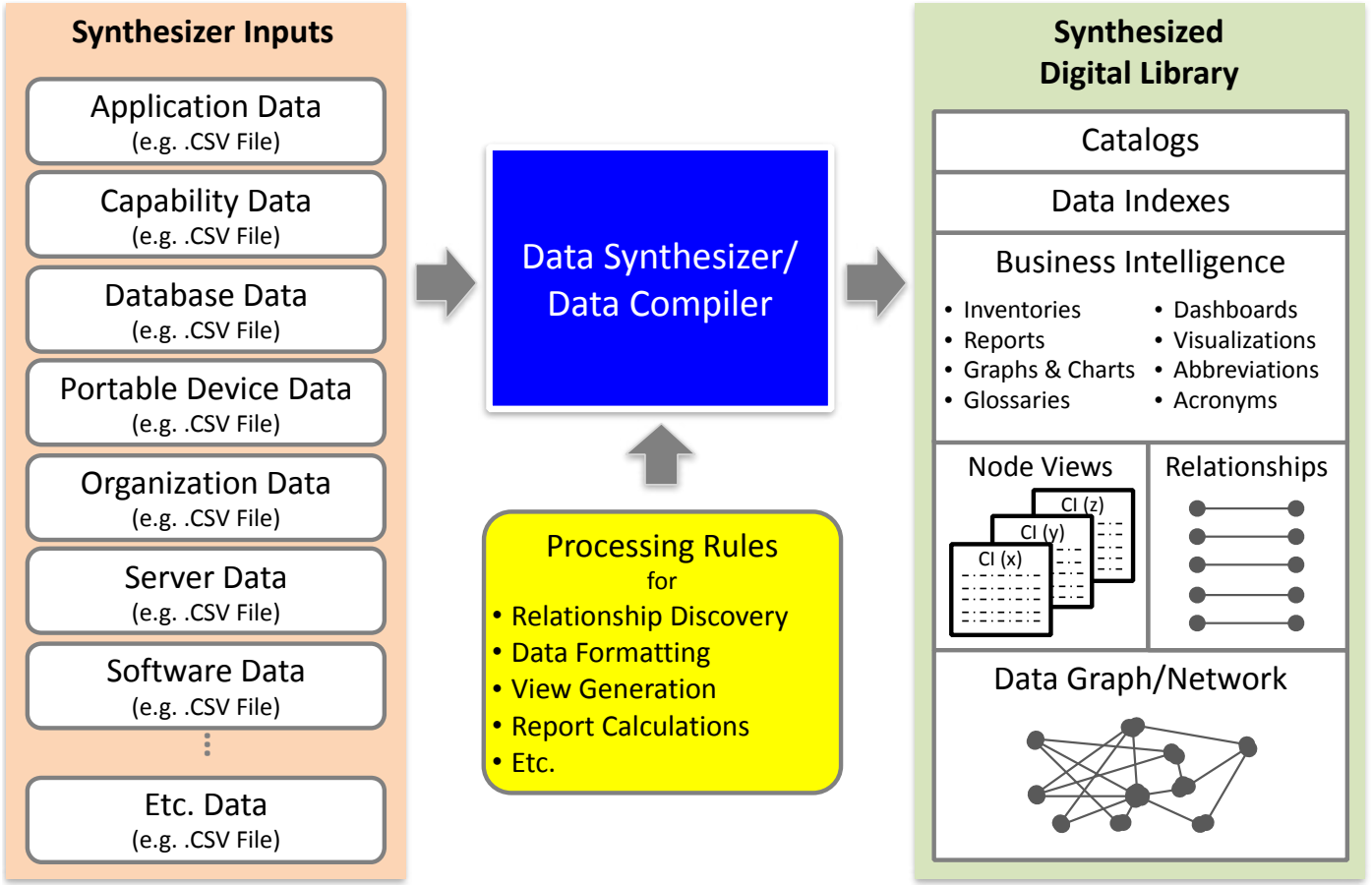


Figure 3. Conceptual architecture for NOUNZ data-driven synthesis.

structures that facilitate stronger knowledge management. In other words, they help end users more effectively and efficiently explore, discover, learn, and understand. Examples of KCs include but are not limited to navigation and classification taxonomies, library catalogs and indexes, simple charts and graphs, semantic data graphs and relationships, sortable lists and tables, reports, interactive dashboards, and complex data visualizations (see Figure 4).

The reader will note that, even with tens of millions of contributors, Wikipedia has very few consistent knowledge constructs and almost none of them are driven by data. In Wikipedia, this means that knowledge structures do not change when data changes, thus requiring human labor to address such changes. For example, list and table structures exist but are formatted differently, from topic to topic. Charts and graphs are inconsistent static images that are first generated in other tools, exported to stand-alone files, uploaded into Wikipedia, and then linked into page content as references to files that cannot change as the data changes below them. This means that the effort necessary to create and manage such knowledge constructs, especially as data changes, is far too great, even for the vast Wikipedia community. On the other hand, data compilers make this type of work easy, quick, and very consistent.

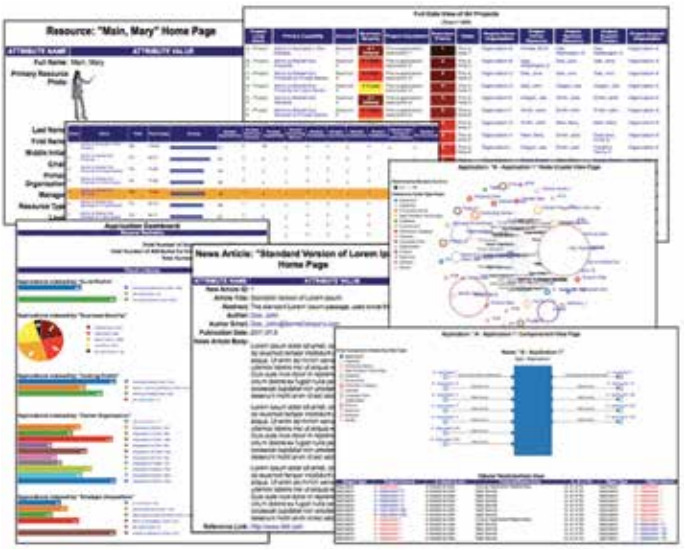


Figure 4. Collage of synthesized knowledge construct examples.

Data compilers allow technical communicators to generate millions of charts, graphs, dashboards, and interactive data visualizations, with nothing more than a few commands that tell the compiler how to handle and format your data. More importantly, data compilers make

it very easy to update all the knowledge constructs when data changes. This means that maintaining and changing millions of charts, graphs, dashboards, and interactive data visualizations becomes much simpler work. It also translates to a very powerful exploration, discovery, and learning experiences for TC end users.

Synthesis of highly curated digital libraries for technical communication

Most technical communicators who work with complex documentation understand the complexities associated with structure development. For example, it takes significant effort to manage document branding, page headers and footers, page sequencing, chapter structuring, bibliographies, index design, figure placements, table of contents generation, etc. All too often, content management websites like enterprise intranets have very little long-term planning and design that go beyond basic layout templates and consistent color schemes. Wikipedia, a far larger and more complex example, had almost no formal planning and design, making it virtually impossible to replicate via manual effort. This is where data compilers that leverage DDS can be very powerful tools for technical communicators.

In addition to automatically generating very large volumes of digital content pages and many different knowledge constructs, data compilers will also perform the work of weaving them all together into consistent and brand-able digital documentation packages that can be deployed to people and systems. This weaving occurs via the automatic generation of vast numbers of HTML links, each with specific structural intent, that facilitate navigation within and across data, content pages, and knowledge structures.

The importance of HTML link generation and maintenance should not be taken lightly. Wikipedia has so many HTML links that many millions of them are considered to be dead links, in spite of having tens of millions of content contributors. The problem manifests itself when underlying data or content changes and documenters must return to identify and update HTML links that are impacted because of the changes. If Wikipedia cannot properly address this problem with manual labor, even with tens of millions of editors, how can a small team of technical communicators address such impacts in their own electronic documentation?

Software like Wikimedia's Wiki, Adobe Technical Communication Suite, and MadCap Flare all provide

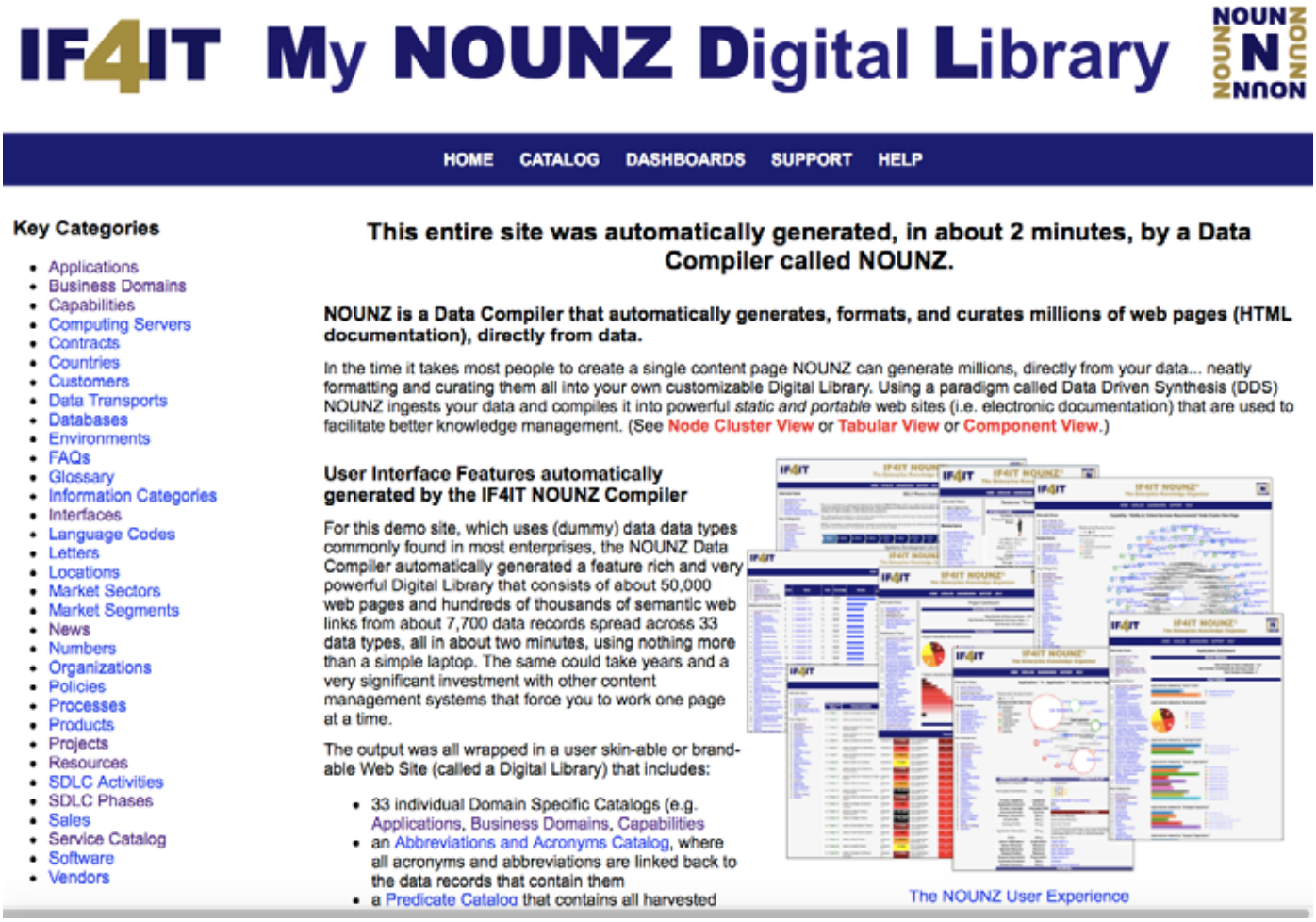

IF4IT My NOUNZ Digital Library
HOME CATALOG DASHBOARDS SUPPORT HELP
Key Categories
Applications
Business Domains
Capabilities
Computing Servers
Contracts
Countries
Customers
Data Transports
Databases
Environments
FAQs
Glossary
Information Categories
Interfaces
Language Codes
Letters
Locations
Market Sectors
Market Segments
News
Numbers
Organizations
Policies
Processes
Products
Projects
Resources
SDLC Activities
SDLC Phases
Sales
Service Catalog
Software
Vendors
This entire site was automatically generated, in about 2 minutes, by a Data Compiler called NOUNZ.
NOUNZ is a Data Compiler that automatically generates, formats, and curates millions of web pages (HTML documentation), directly from data.
In the time it takes most people to create a single content page NOUNZ can generate millions, directly from your data... neatly formatting and curating them all into your own customizable Digital Library. Using a paradigm called Data Driven Synthesis (DDS) NOUNZ ingests your data and compiles it into powerful static and portable web sites (i.e. electronic documentation) that are used to facilitate better knowledge management. (See Node Cluster View or Tabular View or Component View.)
User Interface Features automatically generated by the IF4IT NOUNZ Compiler
For this demo site, which uses (dummy) data data types commonly found in most enterprises, the NOUNZ Data Compiler automatically generated a feature rich and very powerful Digital Library that consists of about 50,000 web pages and hundreds of thousands of semantic web links from about 7,700 data records spread across 33 data types, all in about two minutes, using nothing more than a simple laptop. The same could take years and a very significant investment with other content management systems that force you to work one page at a time.
The output was all wrapped in a user skin-able or brand-able Web Site (called a Digital Library) that includes:
33 individual Domain Specific Catalogs (e.g. Applications, Business Domains, Capabilities
an Abbreviations and Acronyms Catalog, where all acronyms and abbreviations are linked back to the data records that contain them
a Predicate Catalog that contains all harvested
The NOUNZ User Experience

Figure 5. Example home page for an automatically generated digital library.

features that help technical communicators define and build their document structures and sets. Data-driven synthesis tools go a few steps further in that they automatically create vast numbers of dynamic HTML links and use these links to assemble large documentation packages. The NOUNZ data compiler, for example, specifically generates portable electronic documentation packages that are called digital libraries, which can easily consist of millions of HTML links (see Figure 5). These digital libraries are deployable websites that follow the structural navigation, storage, and access patterns of traditional brick-and-mortar libraries. If underlying data changes, technical communicators can simply recompile the entire library with very little effort.

The paradigm of DDS even makes it easy for technical communicators to create multiple digital libraries (i.e., different topic instances) for federated topic management and to manage historical versions, since each compiled output is treated as a separate snapshot in time. And, because the outputs of compilers like NOUNZ can also be represented as Semantic Data Graphs (SDGs), technical communicators can even store them to NoSQL repositories for big data analytics.

DDS Adoption Considerations

Working with data-driven synthesis tools can be very productive for technical communicators. However, doing so requires some new ways of thinking and acting. The first thing to consider is that your focus for documentation and publication will shift from developing raw and unstructured content, in Long Form Notation (LFN), to highly structured Name-Value pair content, using Short Form Notation (SFN). This means the bulk of your content will be driven from data and that you will need to spend more time collecting and preparing data than you would on writing narratives.

Another important consideration is your shift in required skills and in how content is authored. With traditional content management systems like Wikis, you can have non-technical people in your end user audience create and publish their own articles, even if curating them becomes difficult. Shifting to DDS means that one person or a small team of people who know how to install, set up, and run data compilers will have responsibilities for the bulk of content creation and curation. End users who want to create content will have to submit their materials to you for integration and curation with the data compiler. For example, others may write policy documents but they will have to hand them off to you for integration into the policy data inventories that you use to feed the data compiler. The upside is that one person using the DDS paradigm with a data compiler will often significantly outperform an army of people who are manually generating content.

Finally, technical communicators must get used to a constant recompilation of content with DDS data compilers. Using traditional CMSs, people submit content and rarely go back to update it (or do so far less than they should). Using DDS and data compilers means that changes in data cause the compiler to update any and all other data and content that is affected by such changes. The upside is that doing so is easy and quick. This allows you to iteratively improve your data and your content, quickly seeing the impacts of your changes with every new compilation. Prepare for some of these changes as you begin to experiment with DDS methods and models.

Conclusion

The data-driven synthesis paradigm is a different way of thinking about and working with your content. Traditional content management paradigms, such as Wikipedia, have popularized the idea that "everyone is an author" and can publish content in your enterprise. This often results in issues with updating and repurposing content, issues with Long Form Notation (LFN), and poorly curated content. DDS is a paradigm that overcomes many of these issues.

The paradigm of synthesis is not new and has been leveraged for decades in many industries for many different reasons. Specifically, DDS for rapid and automatic generation of content is evolving as a critical means for technical communicators to work more efficiently and effectively. In this article we've shown TC practitioners how they can use DDS and data compilers to automatically create large volumes of richly formatted content, synthesize many different knowledge constructs, and generate fully branded and assembled digital libraries.

Choosing to work with DDS tools comes with different tradeoffs that every technical communicator should be aware of. However, the benefits are significant and offer the technical communicator a pathway to faster and higher quality content generation, at lower costs.

JOEL A. KLINE *(*https://www.linkedin.com/in/drjoelkline*), PhD, APR, is a Professor of digital communications at Lebanon Valley College. Joel teaches courses in technology strategy, e-commerce, and entrepreneurship. He is the treasurer of the International Digital Media Arts Association and accredited in public relations by the Public Relations Society of America. Prior to entering academia, he was a principal at a boutique agency and worked in B2B communication in the private sector. Joel consults and researches in the areas of knowledge management, entrepreneurship, and user experience.*

FRANK GUERINO *(*https://www.linkedin.com/in/frankguerino*) is an expert in areas of semantic data theory and Data-Driven Synthesis (DDS). He specializes in the design and application of semantic data compilers that use these paradigms to solve complex knowledge management problem. He currently serves as the chairman for the International Foundation for Information Technology (IF4IT) (*www.if4it.com*), where he helps publish IT industry best practices and helps educate and certify IT professionals. In addition to his current role, Frank has spent almost 30 years as a technologist, leader, and advisor in the semiconductor, financial, pharmaceutical, healthcare, insurance, and government contracting industries.*

An Introduction to Sphinx AND Read the Docs for Technical Writers

By ERIC HOLSCHER

shutterstock.com/ Jacob Lund

TREATING DOCUMENTATION as code is becoming a major theme in the software industry. This is coming from both developers, who are starting to treat documentation as a priority alongside tests and code, and writers, who are seeing a lot of value in integrating more into the development process. This marriage of cultures isn't simple, but having the proper tools for the job makes both sides happy with process and results.

A lot of developer tools don't work well for writers. Sphinx and Read the Docs are unique in this ecosystem in that they have powerful features that both writers and developers want in one convenient package.

Overview of the Ecosystem

Read the Docs is the largest open source documentation hosting site in the world. Open source in this context means that the code is open source and the documentation hosted is for open source software. It is a developer-focused platform, but it has most of the features that technical writers have come to expect in a tool as well. This blending of worlds makes it the best-suited platform for teams that want both writers and developers contributing to product and API documentation.

Read the Docs is best thought of as a continuous documentation platform for Sphinx. Continuous

documentation is analogous to continuous integration of code, which runs the tests on each commit. In practice, this means that your documentation is built, tested, and updated on every commit.

Sphinx provides a documentation generator that is best-in-class for software docs. Sphinx documents are written in the reStructuredText markup language. reStructuredText is similar to Markdown, but much more powerful, as you'll see in this article. Read the Docs provides a hosting platform for Sphinx, running a build on each commit of your repository.

As a writer who uses Sphinx, you write reStructuredText in plain text files. You then build your documentation by running Sphinx on the command line. Generally it's easiest to output HTML for local writing and testing, and then you can let Read the Docs generate PDFs and other formats.

This article provides an overview of the features of Sphinx and Read the Docs, and enables you to evaluate them for use in your organization.

Introduction to Sphinx

Sphinx (*www.sphinx-doc.org*) is the documentation tool of choice for the Python language community, and increasingly also for other programming languages and tools. It was originally created in 2008 to document the Python language itself.

Over the past eight years, it has turned into a robust and mature solution for software documentation. It includes a number of features that writers expect, such as:

Single Source Publishing

- Output to HTML, PDF, ePub, and more
- Content reuse through includes
- Conditional includes based on content type and tags
- Multiple mature HTML themes that provide great user experience on mobile and desktop
- Referencing across pages, documents, and projects
- Index and Glossary support
- Internationalization support

It also has practical and powerful tools for documenting software specifically, including:

- Semantic referencing of software concepts, including classes, functions, programs, variables, etc.
- Including code comments in documentation output for many programming languages
- Tools for documenting HTTP APIs
- Extensions with the Python language
- A vast array of third-party extensions providing powerful new roles and directives

This article isn't enough to cover all of the power packed into this tool. But I hope to pique reader interest so that you can try these tools out and research their capacity for yourself.

Using Sphinx

reStructuredText is a powerful language primarily because the syntax can be extended. reStructuredText supports two types of extension:

- Paragraph level (with Directives)
- Inline level (with Interpreted Text Roles, or roles for short)

Paragraph Level Markup

Let's see a basic example of a *Directive* in reStructuredText:

```
.. warning:: Here be dragons! This topic
covers a number of options that
  might alter your database.

  Proceed with caution!

.. figure:: screenshot-control-panel.jpg
  :width: 50%

  An overview of the admin control panel.
```

Here `warning` acts as the name of the directive, and is the part you can extend for custom directives. In the `figure`, `screenshot-control-panel.jpg` is an *argument*, `:width:` is an *option*, and the rest is the *content*. They enable customization of directives, and show the full power of reStructuredText. You'll notice that Sphinx uses whitespace to denote where a directive ends. The "Proceed with caution!" is still part of the `warning`, and everything that continues to be indented will be part of that warning.

Sphinx ships with a number of powerful directives for documenting code. You can also write your own if you have someone on your team that knows Python.

Inline Markup

For extensibility inside of a paragraph, Sphinx uses roles. Here is an example:

```
You can learn more about this in 1984. It is
implemented in our code at System.Security
```

Here you can see `rfc` and `class` act as the name of the role, and then you can pass in a single argument. For the `rfc` role, it generates a link to the online reference for RFC 1984 with a text of `RFC 1984`.

The `class` role is where things get interesting. This acts as a reference to the class `System.Security` that is documented in your project, which is a hyperlink in the HTML output, but also a working link in PDF and other outputs.

You can implement your own roles with Python. This enables you to create powerful semantic constructs inside the actual markup you're using to write documentation. If you worked building software at a toy factory, you could create a custom semantic sytnax for our own software:

```
Check out :jira:`199` for information on
the :toy:`jump-rope`.
There is a fix in our
:unit-test:`assert-jump-rope-length`.
```

These custom roles work in your `.rst` files, but also in any kind content that is pulled out of a comment in your source code, too.

Let's see how to take these sets of reStructuredText files and turn them into a document set.

Table of Contents (TOC) Tree

Sphinx lets you combine multiple pages into a cohesive hierarchy. The `toctree` directive is a fundamental part of this structure.

```
.. toctree::
  :maxdepth: 2

  install
  support
```

The above example outputs a Table of Contents in the page where it occurs. The `maxdepth` argument tells Sphinx to include two levels of headers. This also tells Sphinx that the other pages are subpages of the current page, creating a "tree" structure of the pages:

```
index
├── install
├── support
```

The TOC Tree is also used for generating the global navigation inside of Sphinx. The index at the top of your project acts as the root of the navigation. The `toctree` is quite important and one of the most powerful concepts in Sphinx. More info on the `toctree` can be found online *http://www.sphinx-doc.org/en/stable/markup/toctree.html*.

Working with Code

Showing code examples is a common task in all documentation. Sphinx is quite well prepared for this task. Let's see how we can show a basic code example:

```
.. code-block:: python
  :linenos:

  import antigravity

  def main():
    antigravity.fly()
```

This markup displays the Python snippet with syntax highlighting and line numbers. The `python` in the above example is an argument to the `code-block` *directive*. The `:linesnos:` is an *option* that says to display line numbers for the code block. In the HTML output, it would look like this:

```
1  import antigravity
2
3  def main():
4      antigravity.fly()
```

Sphinx doesn't stop there though. It allows you to store your code examples in external files and be included in your documentation for easier maintenance. This uses the `literalinclude` directive:

```
.. literalinclude:: example.rb
  :language: ruby
  :emphasize-lines: 12,15-18
```

The neat addition here is the `:emphasize-lines:`. This shows the lines highlighted in your output. This is quite useful for showing sets of code examples where a subset of the code changes. You can also specify just a subset of lines to show with the `:lines:` option, so you don't have to manage multiple snippets.

There's far more power to Sphinx directives than this article can show. You can see the full documentation for them on the Sphinx website, *http://www.sphinx-doc.org/en/stable/markup/code.html*.

Working with References

A powerful reference system is a large part of the power of Sphinx. You are able to reference arbitrary headings and paragraphs in your content, but also semantically reference a large number of programming concepts as well. On top of that, Sphinx includes `intersphinx` which allows referencing across Sphinx projects. This means that if you have multiple projects inside your company, you can still use semantic referencing across them!

A simple reference is defined like this:

```
.. _reference-target-name::

This is a bit of content.

This is how you point to the above
reference, :ref:`reference-target-name`
```

Sphinx also includes a number of other semantic reference types. Examples of other semantic reference types that Sphinx provides:

- `:doc:` for referencing documents
- `:cls:` for referencing programming classes
- `:term:` for referencing glossary terms

All references support `intersphinx`, which allows you to prefix your references with a specific project name. So if your user guide needs to reference your API documentation, you could write, `Check out the`

`:cls:`api-ref:api.request`` which would know to look in your `api-ref` project for the class. The name `api-ref` is arbitrary, and is defined in your project configuration's `intersphinx _ mapping`.

You can see the complete set of references in the Sphinx documentation:

- Cross Referencing Syntax—*www.sphinx-doc.org/en/stable/markup/inline.html#cross-referencing-syntax*
- Intersphinx—*www.sphinx-doc.org/en/stable/ext/intersphinx.html*

Including Comments from Source Code

Integration with code is one of the largest benefits of Sphinx. You can easily embed software comments from multiple languages, including Python, Java, and .NET.

Here is an example of embedding Python documentation for a class in your project:

```
Make a request
--------------

You can make requests using the
:func:`api.request` module.
It makes HTTP requests against our
website.
Here is the basic function signature:

.. autofunction:: api.request
```

As you can see, you can include generated content in the file that you're writing by hand. This allows for building a narrative around generated code comments, instead of giving your users a separate User Guide and API Reference, which is often times just a alphabetical listing of code.

Documentation is best written by humans. Pulling comments from source code is valuable, but it should be a small part of a proper set of documentation. You can read more about including code comments in the following documentation pages:

- Domains, where the references are defined—*www.sphinx-doc.org/en/stable/domains.html#what-is-a-domain*
- Autodoc, which generates docs from Python code—*www.sphinx-doc.org/en/stable/ext/autodoc.html*
- Breathe, which bridges doxygen and Sphinx—*https://breathe.readthedocs.io/en/latest/*
- AutoAPI, which bridges .NET and Sphinx—*https://sphinx-autoapi.readthedocs.io/en/latest/*

Tables

Working with tables can be the bane of anyone who uses plaintext markup languages. Most other languages require that you write them in the file with some arcane syntax. However with reStructuredText, you can use directives to make this much easier.

You can use either of two powerful list directives, `csv-table` and `list-table`. Here is an example of `csv-table`:

```
.. csv-table:: Frozen Delights!
   :header: "Treat", "Quantity",
"Description"
   :widths: 15, 10, 30

   "Albatross", 2.99, "On a stick!"
   "Popcorn", 1.99, "Straight from the oven"
```

This shows the inline markup, however the CSV can also be managed in an external file. This allows you to manage your complex tables in a third-party tool and have your documentation consume them from a CSV, which is a much nicer workflow.

Internationalization

Sphinx includes support for translating documentation into multiple languages. Since Sphinx knows the structure of your documents, it is able to generate translatable strings split by each paragraph, heading, or figure.

Sphinx internationalization works using the *gettext* system. It ships with a builder that allows you to generate a catalog:

```
make gettext
```

You can then use that catalog as the base translation, using any tool that supports gettext. I recommend using Transifex, which gives you a Web-based system for translating the documentation (see Figure 1).

You can then tell Sphinx what language to generate for its documentation when you build it by setting the `language` setting. Read the Docs also supports internationalization, allowing you to host multiple languages of your project documentation.

More information on Sphinx internationalization can be found online at *www.sphinx-doc.org/en/stable/intl.html* and *http://docs.readthedocs.io/en/latest/localization.html*.

My Blog

Sphinx is quite versatile, which means you can use it for a lot of different use cases. I use a package called `ablog` for hosting of my blog over at *http://ericholscher.com*.

The most basic usage allows you to specify that a document is a blog post with the `post` directive:

```
.. post:: 2016-03-15 09:00
   :tags: writing, stc, sphinx
```

The `post` directive adds the document to the set of posts. You can show a post listing by using the `postlist` directive:

```
.. postlist::
   :tags: stc
```

This shows some of the magical things you can do with Sphinx's extensibility. If you're curious, this article was written in reStructuredText and then exported to Word for publishing. You can see the full source online at *https://github.com/ericholscher/ericholscher.com/blob/master/site/blog/2016/jul/1/sphinx-and-rtd-for-writers.rst*.

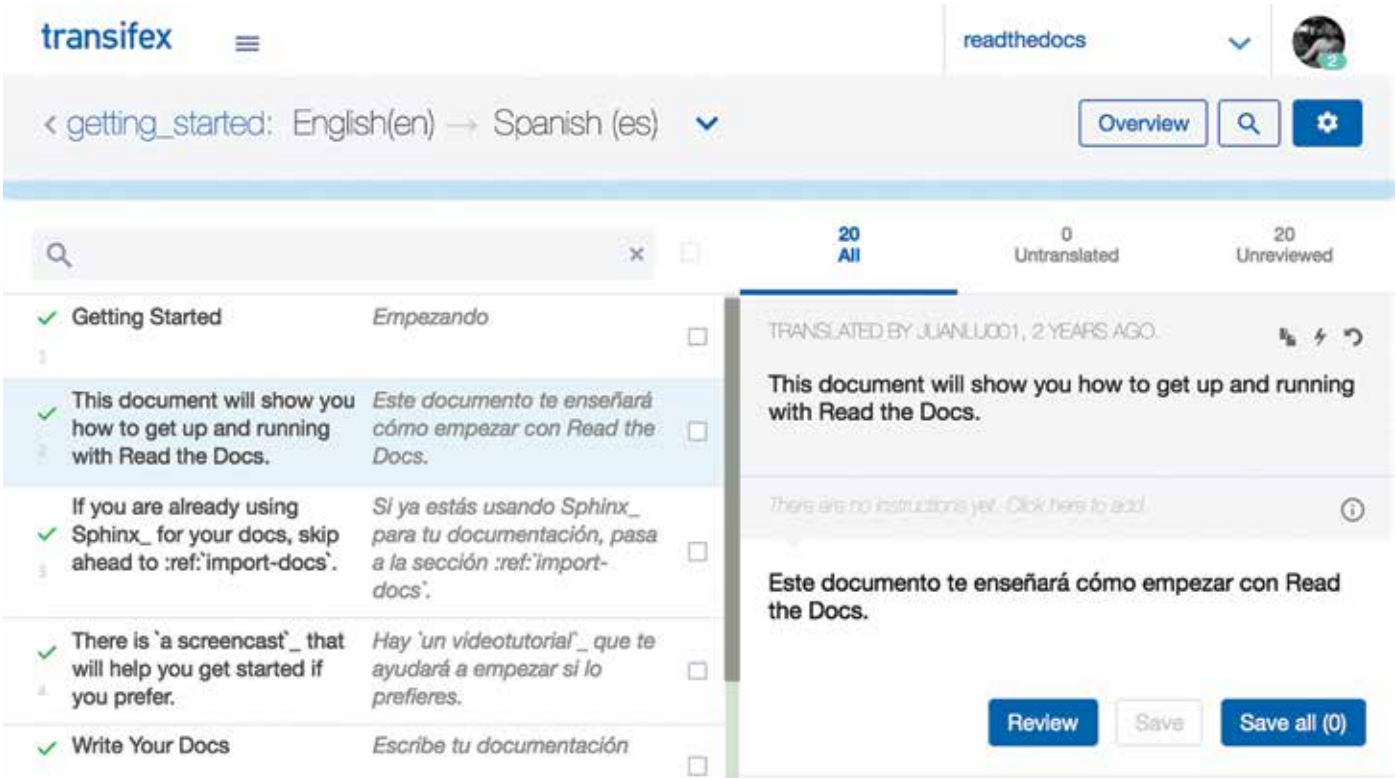

transifex
readthedocs
getting_started: English(en) → Spanish (es)
Overview
20
All
0
Untranslated
20
Unreviewed
Getting Started
Empezando
This document will show you how to get up and running with Read the Docs.
Este documento te enseñará cómo empezar con Read the Docs.
If you are already using Sphinx_ for your docs, skip ahead to :ref:`import-docs`.
Si ya estás usando Sphinx_ para tu documentación, pasa a la sección :ref:`import-docs`.
There is `a screencast`_ that will help you get started if you prefer.
Hay `un videotutorial`_ que te ayudará a empezar si lo prefieres.
Write Your Docs
Escribe tu documentación
TRANSLATED BY JUANLUC01, 2 YEARS AGO
This document will show you how to get up and running with Read the Docs.
There are no instructions yet. Click here to add.
Este documento te enseñará cómo empezar con Read the Docs.
Review
Save
Save all (0)

Figure 1. Transifex

Custom Builders

Sphinx supports custom builders to perform tasks beyond providing basic output formats. Some examples that ship with Sphinx:

- A linkcheck builder that tells you about broken URLs
- A builder that outputs all the changes in the latest version of your code for a changelog
- An XML builder that outputs a representation of your documents in XML
- A Man page builder that builds man pages from your documentation
- JSON builder that outputs your pages as HTML inside of JSON, with some metadata, for embedding dynamically

You can get as creative as you like with custom builders, which is another place to extend Sphinx outside of the markup.

Tradeoffs with Sphinx

Every tool has its issues and limitations. I'd like to address some of the issues with Sphinx, so that you can go into it with eyes wide open.

The biggest issue is that it is originally a programmer tool. This means that a lot of the designs assume knowledge of code, especially for installation and extending the tools. Knowledge of the command line is definitely required.

The markup language, reStructuredText, is also a bit finicky. It depends on whitespace for separation of content, which is a natural concept for Python programmers, but not for most writers.

In general, though, a lot of the complexity in the language comes from the extensibility and power. When compared to something like Markdown, reStructuredText can do so much more that it's worth the complexity and somewhat steep learning curve.

Introduction to Read the Docs

Read the Docs is a hosting platform for Sphinx-generated documentation. It takes the power of Sphinx and adds version control, full-text search, and other useful features. It pulls down code and doc files from Git, Mercurial, or Subversion, then builds and hosts your documentation. We'll use GitHub in this example as it's the most commonly used system for accessing code.

To get started, you create a Read the Docs account, and link your GitHub account. Then you select the GitHub repository you'd like to build documentation for, at which point the magic happens.

Read the Docs will:

- Clone your repository
- Build HTML, PDF, and ePub versions of your documentation from your `master` branch.
- Index your documentation to allow full text search
- Create version objects from each *tag* and *branch* in your repository, allowing you to optionally host those as well

- Activate a webhook on your repository, so when you push code to any branch, your documentation is rebuilt

Whenever you commit new code or documentation to your repository, your documentation is kept up to date. This works with your `master` branch, as well as any other branches or tags you might have activated documentation for.

Read the Docs has two special versions:

- `latest` which maps to the most up-to-date development version of your software
- `stable` which maps to the latest tagged release of your software

These are version aliases that help maintain stable URLs for the most up-to-date commits or for the most stable released version of your software.

We built Read the Docs to be "set it and forget it." Once you set your project up and activate the versions you want hosted, we sit downstream of your version control system and just keep your documentation up to date. It feels pretty magical once it's set up, and it takes the thankless task of deploying documentation out of your day.

Recommended Versioning System

Read the Docs only works with version control. This means that however you version your software, your documentation follows suit. This is great for integrating with your development team, but it also means you need to think about the proper strategy for versioning.

After working with a lot of open source projects, we generally recommend this workflow for projects:

- A `master` branch that the next release of your software is developed on
- Git *branches* for ongoing maintenance of each version of your software that is maintained
- Git *tags* for specific released versions that users might be using

We recommend release branches because it allows you to update them over time. Git *tags* are static, so they are appropriate for specific versions that a user might have installed. An example:

- `master` is your 2.2 release that isn't out yet
- `2.1.X` is your release branch for the `2.1` version
- `2.1.1` is a similar tag of your `2.1` branch, with the latest release
- `2.1.0` is the first tag of your `2.1` branch

Additional Read the Docs Features

Read the Docs also provides the following features:

- GitHub, Bitbucket, and Gitlab webhooks
- Custom domain hosting
- Full-text search for all your projects' versions
- Status badges to show your docs are up to date
- Hosting of multiple languages for a specific project
- Hosting of multiple projects on a single domain with "subprojects"

Feel free to email me or find me at a conference if you want to talk more about these concepts, or if you have ideas for other neat features.

Real Life Examples

Read the Docs has a large number of users across many different programming languages. This is in part because Sphinx is such a powerful and dynamic tool, and Read the Docs makes it easy and free to host docs for your open source project.

Here are some examples that show the wide range of projects using Sphinx and Read the Docs:

- ASP.NET—Microsoft's Web framework, *https://docs.asp.net*
- Julia—A language for scientific computing, *http://docs.julialang.org*
- Jupyter—An interactive programming environment, *http://jupyter.readthedocs.io*
- PHPMyAdmin—A Web-based database editor, *https://docs.phpmyadmin.net*
- Write the Docs—The community site for Write the Docs, *www.writethedocs.org*

As you can see, a wide range of projects are using the tools for many different uses. It's a powerful tool for writing prose as well as documenting source code.

Going Forward

Sphinx is an incredibly powerful tool. Read the Docs builds on top to provide hosting for Sphinx documentation that keeps your docs up to date across versions. Together, they are a wonderful set of tools that developers and technical writers both enjoy using.

I firmly believe that the more integrated with the product development process technical writers get, the better our products get. Tools that integrate documentation and development workflows make it much easier for writers to become part of the larger development process. Sphinx and Read the Docs have been battle tested by hundreds of thousands of open source developers for years, and are a great choice for your software documentation project.

ERIC HOLSCHER (eric.holscher@gmail.com) *is the co-founder of Read the Docs and a co-organizer of the Write the Docs conference. Along with community organizing and open source work, he does consulting around software documentation and speaks at a number of industry events each year. When he isn't expressing his views on software docs, he's getting views from the top of mountains.*

shutterstock.com/ K Kreto

Technical Writing Supported by a Product Lifecycle Management System:

A Better Approach for the Creation of Product Documentation

By CHRISTIAN BRAND

IN THIS ARTICLE you will learn the concept of PLM-supported authoring and single-source publishing, as well as the benefits compared to a stand-alone authoring solution or isolated content management systems.

Technical writing and the creation of quality product documentation is a complex field with many pitfalls that have been eliminated with modern tools. Unstructured authoring and mishandling of files on network drives are issues of the past. A modern Component Content Management System (CCMS) offers the functionality and controls necessary to support the author during the editing and publishing process.

You've just implemented a new CCMS…

Today's systems on the market bring these rich features:

- Database-supported content management
- Separation from content and form (XML authoring)
- An information framework which defines structure, content elements, and rules (DITA, S1000D just to name a few)
- Translation control
- Single-source publishing to different layouts and formats

Strengths and weaknesses are often revealed during the implementation process. The CCMS of your choice most likely comes with its own database, a proprietary file server, needs to be integrated with other systems, has its own user management, has its own support for release processes and workflows, etc. In short, you've just created another island-solution within the growing IT landscape of your company.

Getting down into the details

To further streamline and automate the creation of product documentation, data from other systems is needed. Because the CCMS is not connected to any other systems in the company, you need to consider building interfaces to push, pull, or synchronize data. Once this is done, you'll need to constantly monitor these interfaces to ensure everything is functioning smoothly until the next maintenance window.

Because the CCMS also has its own user management facility, you'll be adding another user credential system to the growing list or you'll have to once again enhance the configuration of your single-sign-on environment.

Consider your document approval workflow. The CCMS is not directly connected to your company's Document Management System (DMS). This often means you'll need to duplicate the output and keep copies in both your CMMS and in your DMS.

Let's summarize

The CCMS, without doubt, supports and enhances your authoring process. Moving from (unstructured) writing in a word processor and managing files without version control to structured authoring in XML and controlling files in a modern content management system is a quantum leap in efficiency and control.

However, decision makers should be familiar with the downsides of a stand-alone system, especially when evaluating a new company-wide solution for technical documentation:

- It increases complexity of existing IT landscapes
- It creates additional costs associated with setup and maintenance, including:
  - Additional servers
  - Additional databases
  - Additional storage space
  - Integrations with other systems
- It creates additional running costs associated with additional systems and tools needed
- It incorporates different processes and workflows which often minimize the new time savings
- It (still) creates redundant content

What would a different approach look like?

To create a new product, a company employs designers and engineers who create CAD models, drawings, bills-of-materials (BOMs), specifications, data sheets, manufacturing plans, and much more, to bring a product to life.

To manage all of this relevant product data, Product Lifecycle Management (PLM) systems are widely used in all different industries. Within the PLM system, everything concerning the product, from initial CAD model requirements to the latest specification and defect protocol, is stored under version control and managed by well-defined processes.

Is everything from the product environment really managed in PLM? What about product documentation? Wouldn't it be beneficial to use the existing PLM system for product documentation? What better place could there be to close the gap between product development and product documentation?

Imagine the benefits that could be realized by managing product documentation in your company's PLM system:

- Reduce the Total Cost of Ownership (TCO) by taking advantage of your existing hardware and software infrastructure

- Manage all product documentation assets in the PLM system
- Access existing PLM information such as BOMs and rendered product images and integrate them in your documents
- Integrate PLM metadata such as revision, author, document number, etc. automatically into your documents
- Use existing PLM processes and workflows to review and approve your documents
- Make product documentation assets easily available to marketing or presales departments
- Close the gap between product development and product documentation

PLM-supported authoring tools are not a theoretical approach that might be realized in the future but exist for almost all major/leading PLM vendors (Arbortext for PTC's Windchill PLM system, XPLM Publisher for Oracle Agile PLM, or TecPub Studio for Siemens Teamcenter).

Let's take a closer look at one of these solutions. XPLM Publisher is the authoring and single-source publishing solution available for Oracle Agile PLM. It provides all of the functionality listed above. XPLM Publisher is a DITA CCMS which is tightly integrated in Oracle Agile PLM, combining the best of both worlds—the power and flexibility from content management built into the PLM system and the professional authoring and publishing part of a standalone CCMS.

Conclusion

Product documentation is a critical part of the product record and is best managed in the system that contains the product record. By consolidating the product record into a single system, the high cost of additional IT silos can be avoided. With this approach, users can rely on standardized processes and controls provided by the PLM system to ensure efficient routing, approval, and control of the entire product record. When considering a CCMS, it is beneficial to first check the alternatives to standalone solutions on the market which integrate into your PLM system. You will be better off if the new solution complements your existing PLM system and will not be a new, redundant tool in your company's IT infrastructure.

CHRISTIAN BRAND *works as projecteer in the area of technical documentation, content management, and PLM systems. He's a trained technical writer with a flair for structures and well-phrased content.*

The Benefits of DITA Authoring WITHIN A Component Content Management System

shutterstock.com/ Flat Design

By KEITH SCHENGILI-ROBERTS

IT IS RARE for documentation teams to not use some form of database or repository for storing their technical content. A documentation team of any size will generate significant amounts of content over time that eventually outgrows shared network folders. This is doubly so for any documentation team that moves to a topic-based writing environment like DITA, as the concept of topic reuse relies upon there being a significant amount of previous content to leverage.

What I have observed in documentation groups over the years is that there often comes a point where any further growth depends on moving away from a simple file folder structure or database to a content management system; and more specifically for DITA-based content, a Component Content Management System (CCMS) where the "components" are individual topics.

When a documentation team is searching for a CCMS, they often do not know about all of the features that typical mature CCMSs have to offer. Moving to a CCMS is an opportunity for a documentation team to reinvent themselves and improve their processes, not to ossify existing and outmoded practices. When crafting a request for purchase of a CCMS, think about where you want your documentation team to be in five years and find a CCMS that can get you there. The following provides a brief survey of the types of features and benefits that can be gained by moving your DITA documentation processes over from a file system to a CCMS.

Versioning of Content

Content versioning is a feature common to most content management systems, although not to most file systems. Originally created for software development, versioning allows users to "check out" content for authoring and "check in" any changes. This practice is for programmers, as it allows them to revert to a previous version of their code to help root out bugs. Similarly, a typical CCMS will have versioning capabilities that ensures that only a single technical writer can work on a given topic at any time, and will also register who made an edit and allow for a quick comparison between versions. Just like a programming environment, any documentation "bugs" that are discovered in a later version of a topic can be reverted to an earlier version without problem. A CCMS should also include the ability for authors to comment on the topic- and map-level changes they are checking into the system so that someone else can tell at a glance what changes occurred and when.

Workflow

Another feature mature CCMSs provide is workflow. In addition to being able to check in/out versioned content, authors can also notify the system when they consider their work on a topic to be complete. Depending on how the workflow is set up, the topic can then be routed automatically for someone else to review and approve, such as an editor or product manager. Automating workflow introduces many ways to improve documentation processes and quality. In an engineering environment, subject matter experts (SMEs) may be assigned within the system to write content, which is then "polished" by a technical writer, and then sent to another SME for approval. This type of workflow implies that roles can also be assigned to contrib-

utors, helping define who can do what types of actions with particular content.

The ability to create workflows also sets up the possibility to schedule deadlines for when work is expected to be completed. A robust CCMS will be able to include not only the final deadline for when all of the topics within a document are due, but also various stages in between in order to help content contributors stay on track, and to notify them when they are falling behind schedule.

CCMS workflow can ensure that all topics within a document have been verified prior to publication, and that internal quality, external requirements—like a medical standard or legal standards—are met. In addition to being able to route content for review, a robust CCMS will have other workflow processes that can be triggered, such as automatically routing the finalized and approved version for localization purposes.

Measurement

Topic-based authoring allows for much more finely-grained measurement of content production. No longer are documentation managers restricted to only measuring when a document is completed and published, but they can now get a better understanding as to how long it takes to produce content. This is critical to know when planning ahead for future documentation releases. A good CCMS will enable a documentation manager to retrieve a wide range of metrics, allowing them to not only effectively measure the return on investment (ROI) for purchasing a CCMS, but also to understand ongoing production and quality issues. As an example of the latter, it should be possible to measure the topic reuse rate, providing the documentation manager information on whether existing content is being properly leveraged in the production of new content. Other types of production measurements, many of which relate to content quality, can also be measured effectively using a CCMS, including:

- Determining whether or not there is an optimal ratio of topic types being used (e.g., ensuring that there are enough task topics or troubleshooting topics in your user manuals)
- Establishing readability metrics for individual topics
- Tagging consistency throughout your document set
- Checking quality and consistent use of reused content

If a CCMS includes workflow features, it should also be possible to visually chart progress and how much work has been finished for a completed document. This type of function is critical in Agile-based development environments, making it possible for a documentation manager to attend Scrum meetings and show the velocity of documentation progress.

Localization

Localization savings are often a key driver in the move to DITA and a CCMS, so when choosing a system, you want assurance that it manages localization well. Content reuse in the source language equates roughly to localization reuse, guaranteeing significant cost savings from the start. But there are other key features that can help further maximize your localization budget. First, ensure that it works natively with Unicode character sets, as this reduces costs that could otherwise be incurred from having to deal with code-page issues when a character in one language is displayed as another. Confirm that the localization process is easy to use, so that packaging up content to be translated and importing the localized content back into the system can be handled by someone who may not necessarily be skilled in translation. In my experience, it is rare for a firm to have dedicated localization coordinators whose sole job is to handle this process, so it needs to be simple and straightforward while also providing tools for "power users" who can verify the quality of the translation. Look for flexibility in translation formats: DITA and XLIFF as packaging formats for translation firms to work with, and the ability to handle other XML formats such as MathML and SVG easily.

Similarly, beware of CCMSs whose processes are based on proprietary localization formats. This helps avoid the possibility of vendor lock-in, and any process that transforms your XML to another format and back is error prone. A CCMS-based localization process should ultimately deliver on the promise of cost savings and be easy to use and robust enough to grow with your localization needs over time.

Summing Up

A typical commercial CCMS will include most of the features mentioned here and more. While it is possible to use DITA within a small team using a file folder filled with topics, documentation teams really begin to shine when they have a CCMS optimized for use with DITA.

Read CCMS case studies, go to online DITA forums (such as those on LinkedIn and Yahoo!), and visit vendor websites to get a better understanding of what is available. There are several hundred firms using DITA worldwide, so reach out to your colleagues and learn what they have to say and what they recommend. And when selecting a CCMS, do not focus strictly on cool functions, the UI, or even the price of the system; instead think of where you need to take your documentation team and what your users will expect of you five years from now.

KEITH SCHENGILI-ROBERTS *is a DITA specialist at IXIASOFT, a DITA evangelist, and a technical writing nerd. Keith is an award-winning lecturer on information architecture at the University of Toronto's School of Continuing Studies. He is an active member of the OASIS DITA Technical Committee and can often be found presenting at conferences, working with customers, researching how DITA is being used, and sharing those results with the DITA community. Keith's popular industry blog* DITAWriter.com *has become a focal point on DITA resources and best practices. Connect with Keith on Twitter* @KeithIXIASOFT.

2017 Membership Season Now Open

BE PREPARED FOR 2017 when you renew your STC Classic membership starting 15 September and save $30 on 2017 dues. This early renewal discount brings your membership rate down from $225 to $195, but you must act quickly to receive the discount. Please enter promo code **STC2017** in the "Dues Discount" field at the end of the online application.

STC is committed to consistently adding value for members. STC membership provides exclusive member pricing on education, certification, the STC Summit, and more—yet another return on your membership investment. You will find that these savings go far beyond the cost of your STC membership. Continue to receive STC's online, award-winning *Intercom* magazine, *Technical Communication* journal, TCBOK, and *Salary Database*; your personalized education report card; Job Bank access; and other networking opportunities from across the globe.

Also it's the perfect time to think about taking advantage of the Gold Value Package member and receive Classic member benefits, plus:

- Five Live Webinars at no cost **($295 value)**
- Additional 20% off all Online Courses **($519+ value)**
- Print subscription to 10 issues of *Intercom* magazine **($60+ value)**
- Early Bird rate on annual conference registration any time of the year **($400+ value)**
- Membership in a local chapter and all SIGs **($145+ value)**

Renew your STC membership today and be a part of *your* Society. Visit *www.stc.org/membership/join-or-renew-now/renew-today* for more information, or email *membership@stc.org* with any questions.

We look forward to welcoming you back as a 2017 member.

Reminder of Deadlines for Awards and Honors

THE DEADLINES for nominations for many of STC's awards and honors are upcoming. Please see the STC website, *www.stc.org*, for more information or to find out how to nominate someone.

- Associate Fellow Recommendations: 15 October
- Fellow Nominations: 1 October
- Sigma Tau Chi and Alpha Sigma Honors Societies: 30 November
- Jay R. Gould Award for Excellence in Teaching Technical Communication: 15 October
- Ken Rainey Award for Excellence in Research: 4 November
- Distinguished Community Service Awards: 25 October 2016
- Community Achievement Awards: 29 January 2017
- Community Pacesetter Awards: 24 March 2017

Share Yourself with *Intercom* Readers

WE'RE LOOKING FOR members to contribute a first-person column for a future issue of *Intercom*. The magazine has a trio of member-focused columns: My Job (your day-to-day work), Off Hours (discussing your hobby or side gig), and Looking Back. My Job is a first-hand account of the day-to-day work of an STC member and what makes their job interesting, fun, or unique. Off Hours is a look at the side jobs and hobbies our members have. And Looking Back focuses on Senior members providing perspectives learned throughout their careers. Would you like to share your story with *Intercom* readers? Email James Cameron, *james.cameron@stc.org*, for more information, samples, and to volunteer!

STC Communities and Staff Win APEX Awards

STC IS PROUD to announce that five STC communities and the STC staff recently were named winners in APEX 2016, the 28th Annual Awards for Publication Excellence. APEX Awards are based on excellence in graphic design, editorial content, and the ability to achieve overall communication excellence. APEX Awards of Excellence recognize exceptional entries in each of the individual categories.

Congratulations to the following STC winners:

- Northeast Ohio Chapter: APEX Award of Excellence in the category of Newsletters—Electronic and Email
- Chicago Chapter (Linda Kelley): APEX Award of Excellence in the category of Newsletters—Electronic and Email
- Carolina Chapter (Lindsey Saunders): APEX Award of Excellence in the category of Newsletters—Electronic and Email
- Technical Editing SIG (Rick Sapir): APEX Award of Excellence in the category of One to Two Person-Produced Websites
- *Intercom* (Liz Pohland): APEX Award of Excellence in the category of Magazine, Journal, & Tabloids Writing.
- *2014–2015 STC Salary Database*: APEX Award of Excellence in the category of One-of-a-Kind Publications.

Congratulations again to the winners! For more information on the APEX Awards and a full listing of winners, visit *www.apexawards.com/*.

Handy GitHub Resources

shutterstock.com/ GaudiLab

BY NICKY BLEIEL | *STC Associate Fellow*

GITHUB IS A Web-based service that hosts code and other content in Git repositories and provides version control, project management, and social features.

There are a variety of resources available to learn about GitHub, including quick reference guides, tutorials, videos, books, and webinars. Create a free GitHub account (*https://help.github.com/categories/setup/*) and take a look.

Learning GitHub

The GitHub Help is excellent. I recommend you read the GitHub Glossary before starting any tutorials to familiarize yourself with the terminology. The GitHub Guides are short (< 10 min) tutorials, and include a "Hello World" project—always a good place to start.

- GitHub Help, *https://help.github.com*
- GitHub Glossary, *https://help.github.com/articles/github-glossary*
- GitHub Guides, *https://guides.github.com*
- Video GitHub Training and Guides, *https://www.youtube.com/c/githubguides*
- GitHub Services (try the free 15 minute "Learn the basics of Git in 15 minutes" tutorial), *https://services.github.com/*. Also sign up for a free introductory webinar, *https://services.github.com/training*
- GitHub Workflow, *https://help.github.com/articles/what-is-a-good-git-workflow*
- Guide to Contributing to Open Source Projects, *https://guides.github.com/activities/contributing-to-open-source*

Mastering Markdown

The GitHub READMEs, comments, wikis, Gists, and more are written in Markdown, a lightweight markup language that can be converted to HTML easily.

- Markdown Quick Reference Guide: "Mastering Markdown" from GitHub Guides, *https://guides.github.com/features/mastering-markdown*

Books

- Bell, Peter, and Brent Beer. Introducing GitHub. Sebastopol, CA: O'Reilly Media, 2014. *http://shop.oreilly.com/product/0636920033059.do*
- Chacon, Scott, and Ben Straub. *Pro Git*. Apress, 2014. *https://git-scm.com/book/en/v2*

Projects of Interest

GitHub hosts many projects that may be of interest to technical communicators. For example:

- Dynamic Information Model (an implementation of an intelligent style guide) project by oXygen XML editor and Comtech Services, *https://github.com/oxygenxml/dim.*
- W3C HTML Specification, *https://github.com/w3c/html*
- DITA Open Toolkit, *https://github.com/dita-ot/dita-ot*
- WordPress, *https://github.com/WordPress/WordPress*
- Typography Handbook by Kenneth Wang, *https://github.com/kennethwang14/TypographyHandbook*

Fun Stuff

The Octodex includes the original Octocat and variations created by the community. The Octocat is the official GitHub Mascot and was created by Simon Oxley, the same artist who created the original Twitter bird.

GitHub supports Emoji; the Emoji Cheat Sheet will introduce you to a variety of emoji you can use in your Readmes, comments, and more. The "Ship It Squirrel" (:shipit:) is used in GitHub comments to indicate that you believe a change is ready to ship. And where else can you read the Apollo 11 source code?

- GitHub Octodex, *https://octodex.github.com/*
- Andersson, Arvid. *Emoji Cheat Sheet*, *www.emoji-cheat-sheet.com*
- Apollo 11 Guidance Computer (AGC) source code, *https://github.com/chrislgarry/Apollo-11/*. (For more information, *http://qz.com/726338/the-code-that-took-america-to-the-moon-was-just-published-to-github-and-its-like-a-1960s-time-capsule/*.)

This column is a quick reference guide exploring "essential" technical communication topics. Please send questions or comments to Nicky Bleiel at *nableiel@us.ibm.com*. Follow her on Twitter: *@nickybleiel*.

Reducing the Cost of Compliance with Intelligent Content

BY MARK LEWIS | *STC Associate Fellow*

I'D LIKE TO SHARE research and findings that I presented at the Intelligent Content Conference for Life Sciences 2014 (*www.slideshare.net/hyperwriters/the-roi-of-intelligent-content-for-life-sciences*). While trying to pick a presentation topic, it occurred to me that I had not heard many discussions about how compliance-related tasks could benefit from the content reuse capabilities of XML-based intelligent content. That is the focal point of my presentation and this article.

Procedures are a common content type that is familiar to life sciences and technical writers, so let's use the procedure content type for this discussion. The processes and cost models in this column are examples that are meant to be customized to match the roles and tasks in the content processes in your organization. I hope they give you a jumpstart. Let's:

- Design a cost model for a basic procedure without content reuse
- Design a cost model for a basic procedure with content reuse
- Design a cost model for a structured authoring process without content reuse
- Design a cost model for a structured authoring process with content reuse
- Compare the cost models and determine the savings with reuse

In the content world, metrics can be used to prove a business case, communicate value, or predict costs and savings. This column explores new thoughts and ideas in the area of metrics and ROI. Please send your ideas, comments, and questions to *mlewis@ditametrics.com.*

Design a cost model for a basic procedure without content reuse

Let's design a cost model for a basic procedure without reuse. Figure 1 shows the content elements that are typically included in a procedure.

Title
Decontaminating Tools and Equipment
Short Description
The Decontaminate feature safely and thoroughly decontaminates tools and equipment.
Pre-requisites
The Mini-Lab 2000 can decontaminate a wide variety of material types. Refer to Materials that can be decontaminated to confirm the tool material is supported.
Context Info
To decontaminate tools and equipment:
Steps
1. Place the tools and equipment on a decontamination tray.
2. Place the decontamination tray inside of the Lab ensuring that no items touch the inner walls of the Lab.
3. Select the Material Type.
4. If the material requires silonaria treatment, set the Silonaria switch to On.
5. Press Start.
6. When the timer expires, open the lab and examine the tools and equipment. If slobophia is observed, then return the item to the Lab and repeat the decontamination process.
Result
The tools and equipment are decontaminated and ready for use in a new procedure.

Figure 1. Typical content elements included in a basic procedure

The cost model should represent the average cost for an author to create a procedure. Cost is expressed in labor hours. The first step is to create a table that includes the content elements in a procedure (see Table 1). For each content element, include the average number of occurrences of that element in a procedure. Next, include the average amount of time required to author each element. The amounts must reflect writing all content with no content reuse. To customize the models, you need to review a sample set of your organization's procedures and determine your own averages.

Cost of a Procedure			
Elements	Occurrences	Cost per Element (hrs.)	Total (hrs.)
Short Description	1	1	1.00
Prerequisites	0	0.25	0.00
Context Information	1	0.25	0.25
Steps	6	0.50	3.00
Steps screen shots	0	0.50	0.00
Task Result	1	0.20	0.20
Task Result screen shots	0	0.50	0.00
Post-Requisites	0	0.25	0.00
Indexing	0	0.20	0.00
Total			4.45

Table 1. Cost of creating a procedure without content reuse

Let's calculate the average cost to create a procedure without reuse. The calculation is:
Cost of procedure without content reuse = Summation of ((average number of occurrences of an element) x (average cost per element)) = 4.45 hours

Design a cost model for a basic procedure with content reuse

To determine the cost of a basic procedure with content reuse, we need a value for the average percent reuse. However, the process for determining an accurate percent reuse involves many steps, but because of the limited length of a magazine article, we have to simplify that process and use a rule-of-thumb reuse percentage of 25%. For details on that process, see the "Content Reuse" chapter of my book, *DITA Metrics 101 (www.ditametrics.com).*

Let's calculate the average cost of authoring a procedure with reuse. We already know the savings is 25%, but we'll need the value in labor hours later in this discussion.

The calculation is:
Cost of a procedure with reuse = ((cost of a procedure without reuse) x (1.0 – percent reuse)) =
(4.45 x (1.0 – 0.25)) = 4.45 x 0.75 = 3.34 hours

We can use this same formula to determine the cost for other roles involved in the content lifecycle.

Design a cost model for a structured authoring process without content reuse

Next, let's look at a structured authoring process that does not include content reuse. When I teach this concept, I design a process with an author, technical reviewer, and editor. But this discussion is about compliance, so let's replace the technical reviewer's role with a compliance reviewer (see Figure 2). Your situation may be different, so customize the roles and tasks to represent the process in your organization.

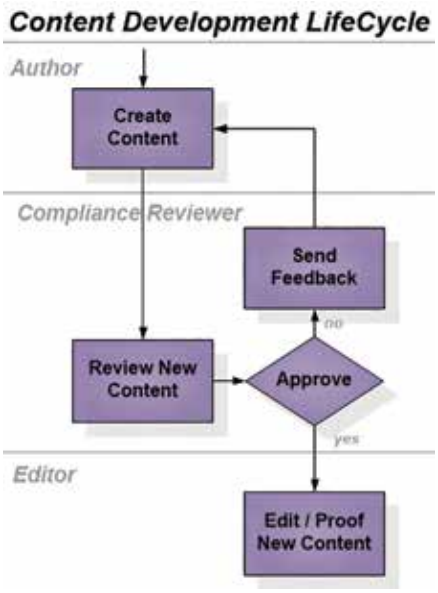


Figure 2. A structured authoring process without content reuse

The equivalent cost model represents the average cost of the process to create a procedure for all roles involved. The first step is to create a table that includes each role and the tasks they perform in the process (see Table 2). Next, include the average amount of time to perform each task. The time estimates are based on the basic procedure which contains an average number of each content element and an average amount of content in each of those elements.

Procedure		
Role	**Task**	**Cost (hrs.)**
Author		
	Create Content	4.45
Compliance Reviewer		
	Review New Content	0.75
Editor		
	Proof New Content	0.33
Total		5.53

Table 2. Cost of the structured authoring process without content reuse

Let's calculate the cost of the structured authoring process without reuse.

The values are:

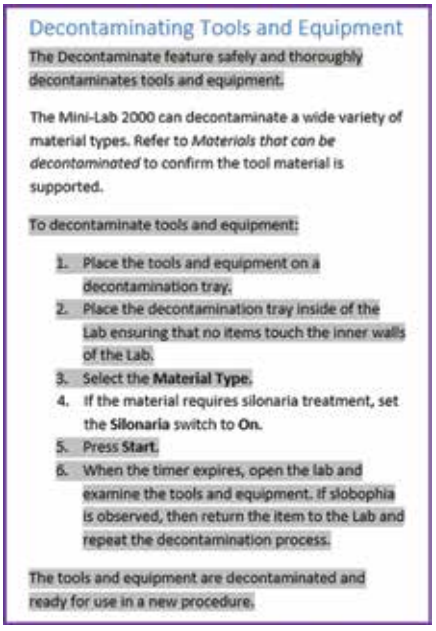
- Cost to create new content for a basic procedure = 4.45 hours from Table 2
- Cost to compliance review new content for a basic procedure = 0.75 hours
- Cost to editor proof new content for a basic procedure = 0.33 hours

The calculation is:
Cost of the structured authoring process without content reuse = (cost to create new content) + (cost to compliance review new content) + (cost to editor proof new content) = 4.45 + 0.75 + 0.33 = 5.53 hours

Design a cost model for a structured authoring process with content reuse

Let's now look at a structured authoring process that includes content reuse. First, let's discuss how the process is different for each role when reusable content is included. When the author is writing the procedure, they avoid writing content from scratch whenever possible. The author searches for and inserts reusable content. The remaining content that is needed must be written by the author and is treated as new content.

In many structured authoring tools, the system shades the background of reused (referenced) content (see Figure 3). This makes it easy for the user to distinguish new content from reused content. This is important for the compliance reviewer and editor, because they know that the content with a shaded background has already been reviewed and proofed and they only need to perform what is known as a *contextual review* of the reused content. A contextual review determines if that content is appropriate or fits in the context of the other content. Performing a contextual review of vetted reusable content takes less effort than a full review or proof of new content (see Figure 4). This is where the biggest savings occur. In our example, this reduces the efforts on new content by the compliance reviewer and editor by the rule-of-thumb reuse percentage of 25%.


Decontaminating Tools and Equipment

The Decontaminate feature safely and thoroughly decontaminates tools and equipment.

The Mini-Lab 2000 can decontaminate a wide variety of material types. Refer to *Materials that can be decontaminated* to confirm the tool material is supported.

To decontaminate tools and equipment:

1. Place the tools and equipment on a decontamination tray.
2. Place the decontamination tray inside of the Lab ensuring that no items touch the inner walls of the Lab.
3. Select the **Material Type**.
4. If the material requires silonaria treatment, set the **Silonaria** switch to **On**.
5. Press **Start**.
6. When the timer expires, open the lab and examine the tools and equipment. If slobophia is observed, then return the item to the Lab and repeat the decontamination process.

The tools and equipment are decontaminated and ready for use in a new procedure.

Figure 3. Reused content has a shaded background

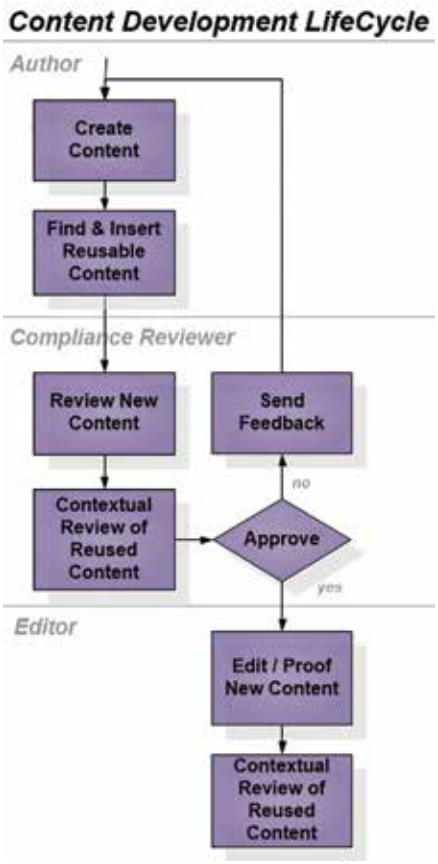


Figure 4. A structured authoring process with content reuse and contextual reviews

The equivalent cost model represents the average cost of the process to create a procedure for all roles involved. First, create a table that includes each role and the tasks they perform in the process. Next, include the average amount of time to perform each task. Remember that this is being performed on the "basic procedure" (see Table 3).

Let's calculate the cost of the structured authoring process with content reuse. The values are:

- Rule-of-thumb reuse percentage = 25% = 0.25
- Cost to find and insert reusable content = 0.11 hours
- Cost of compliance contextual review of reused content = 0.10 hours
- Cost of editor contextual review of reused content = 0.08 hours

The calculations are:

- **A** = Cost to create new content with reuse = (cost to create content without reuse) x (1.0 - rule-of-thumb reuse percentage) = 4.45 x (1.0 - 0.25) = 4.45 x 0.75 = 3.34 hours
- **B** = Cost for compliance review of new content with reuse = (cost to review new content without reuse) x (1.0 - rule-of-thumb reuse percentage) = 0.75 x (1.0 - 0.25) = 0.75 x 0.75 = 0.56 hours
- **C** = Cost to proof new content with reuse = (cost to review new content without reuse) x (1.0 - rule-of-thumb reuse percentage) = 0.33 x (1.0 - 0.25) = 0.33 x 0.75 = 0.25 hours
- Cost of the structured authoring process with content reuse = **A** + (cost to find and insert reusable content) + **B** + (cost of compliance contextual review of reused content) + **C** + (cost of editor contextual review of reused content) = 3.34 + 0.11 + 0.56 + 0.10 + 0.25 + 0.08 = 4.44 hours

Compare the cost models and determine the savings with reuse

For the structured authoring process as a whole, incorporating content reuse reduced the cost from 5.53 hours from Table 3 to 4.44 hours. This is an overall savings of 19.7%.

For the compliance reviewer, incorporating content reuse reduced their effort from 0.75 hours from Table 2 to 0.66 hours from Table 3. This is a savings of 12.0%.

Even though we've simplified some of the calculations by using a rule-of-thumb percent reuse, the results still show that significant savings are possible for all roles involved in the process.

Conclusion

In this article and in my book, *DITA Metrics 101 (http://www.ditametrics.com/)*, I prove the savings possible when intelligent content and content reuse are applied to the content development lifecycle. I show how to design cost models for a structured authoring process with and without content reuse. The difference in cost is the savings,

Procedure			
Role	Task	Subtask	Cost (hrs.)
Author	Create Content		
		Create Content	3.34
		Find and Insert Reusable Content	0.11
		Subtotal	3.45
Compliance Reviewer	Compliance Review		
		Review New Content	0.56
		Contextual Review of Reused Content	0.10
		Subtotal	0.66
Editor	Proof Content		
		Proof Content	0.25
		Contextual Review of Reused Content	0.08
		Subtotal	0.33
Total			4.44

Table 3. Cost of the structured authoring process with reuse and contextual review

which is substantial in all cases. The models that I discuss here are intended to give you a jumpstart and are meant to be customized to match the roles and tasks in the content processes in your organization. In the book, the example process involves the following roles: author, technical reviewer, and editor. In my life sciences presentation and in this article, I customized the example process to include a compliance reviewer. We proved how each role could benefit from the content reuse capabilities of intelligent content. The efforts of many tasks are reduced when content is reused. The compliance reviewer's savings is 12.0%. The overall savings for the structured authoring process is 19.7%. These savings are calculated based on a rule-of-thumb content reuse percentage of 25%. The higher the percent reuse, the greater the overall savings.

The processes in your organization may include additional roles, but I hope you can see how easy it is to add these roles and tasks to the models in this article. I encourage you to take these example processes and metrics and customize them to your organization to prove the savings that you could achieve. Please share your results with the community.

TechComm Standards: Why They Matter, and Where They Don't

shutterstock.com/ Kenishirotie

BY ANDY MCDONALD | *Guest Columnist*

Editor's Note: This month's guest columnist, Andy McDonald, makes us think ahead to new technologies that affect our work. He raises the issue of differences between standards and formats (is DITA one or the other?), and reminds us that STC and other representative bodies need to be involved in the creation of evolving standards or someone else will do it for us.

LET'S TRY TO START with acceptable definitions. And while doing so, let's adopt a business angle. According to *BusinessDictionary.com*, in business:

- A **standard** is seen as a "written definition, limit, or rule, approved and monitored for compliance by an authoritative agency or professional or recognized body as a minimum acceptable benchmark."
- A **format** is seen as a "Communication design: Size, shape, layout, typography, and arrangement of information given or sought in an ad, document, or form, whether prepared for display, printing, or storage."
- A **file format** is seen as a "specific structure or arrangement of data code stored as a computer file. A file format tells the computer how to display, print, and process, and save the data."
- **Information** is seen as "Data that is (1) accurate and timely, (2) specific and organized for a purpose, (3) presented within a context that gives it meaning and relevance, and (4) can lead to an increase in understanding and decrease in uncertainty."

Standard Deviation is a column all about standards—a subject that affects most of our lives, but that we seldom think about. As the title implies, I want to keep the conversation lively and engaging. I'm always looking for guest columnists, and we welcome feedback with comments or requests for standards-related topics to cover. Email the column editor, Ray Gallon, at *rgallon4stc@culturecom.net.*

Additionally, TechWhirl (*http://techwhirl.com/what-is-technical-communications)* defines technical communication as "a field within business communications [that] encompasses a range of disciplines that work together to communicate complex information to those who need it to accomplish a defined task or goal."

Give these definitions, I believe we can agree that, in the tech comm industry, what we do is to manage information and make it available through standards and formats.

What's our purpose?

Purpose is determined by the industry sector in which we work. The aeronautics, chemical, and medical industries are affected by

YOUR RELIABLE PARTNER FOR
CHINESE TRANSLATION!
Simplified & Traditional Chinese
Qualified Resource
High Translation Quality
Competitive Price
ISO 17100 Certified
• 3-Step Translation (Translation+Editing+Proofreading)
• In-house lingustic QA
• Functional QA
• Language Sign-off
• Language Quality Evaluation
• DTP Services
hansemEUG, China
Room 915, Block A, Focus Square, No.6 Futong East Street, Chaoyang District, Beijing
info@ezuserguide.com | www.hansemeug.com
TOP 100 Global 2015

regulations that require practitioners to adhere more to certain standards than to formats. In software, the emphasis is shifting to voluntary standards for improving user experience (UX) quality. Our purpose will change at different paces, depending on whom we work for.

What have we done?

Formats and file formats are often confused with standards. Information is trapped inside these formats. By doing this, we:

- modify how we work, spending more time and effort on the format rather than on the information itself;
- invest a fortune on formats, and by doing so, lose sight of where industry is taking us;
- let the format define the context; and
- spend less time on pertinence, relevance, and decreasing uncertainty, which is probably our major purpose.

What's happening next?

Industry 4.0 will have a major impact on how we deal with information and how it is made available. For example, the Internet of Things has specific requirements for lean contextualized information. And artificial intelligence will have an even greater impact on access and use of information.

Real-time information

More and more, we need to consider real-time information *availability*, not delivery. What role do standards and formats play in this?

- The standards should concern *pertinence* and *context*.
- The formats will be decided by how information is *accessed*, not how it's delivered. Intelligent UI (user interfaces) will tend to impose UX improvements.

Our roles and methods will change in this situation. Standards for information design and curation need to be defined for our future industry.

Delivery, as we know it, is disappearing, and this is what our previous effort on formats was chiefly about. We spent so much time on the latter, that we forgot to look over the hill.

Information has to be readily available (as opposed to made available) as a candidate for collation into contexts that will not be decided by us, but by requirements mapped onto a taxonomy-driven content collection, or even by robots and intelligent agents. Candidates will be collated based on standards that need to be written.

Unless we do this, regulatory agencies will make requirements that may not encompass the changes they do not understand very well (typically, requiring a complete paper manual, even when it is not read).

Emphasis on curation

Making information candidates available is about redesigning some of our methods. This means measuring pertinence and validating it, distinguishing knowledge, know-how, and accepted practices.

Conclusion

We must separate information from its format and start thinking about how to render it accurately, specificly, contextualized, and timely. The standards for information curation in Industry 4.0 need to be defined, and the time to do that is now.

ANDY MCDONALD *is trained as a social psychologist. He has been working in the oil industry since 1988, and is involved in major software documentation projects (CGG & Total). He is now a product manager for innovative information solutions at TechAdvantage.*

MASTER OF SCIENCE
TECHNICAL COMMUNICATION MANAGEMENT
- A unique master's program for the working professional
- Focused on communication and management issues
- Housed in an engineering school
- Taught by faculty with extensive industry experience
- No GRE required
- All online
For more information, contact:
brewer_pe@mercer.edu
478-301-4146
Students may begin in August, January, or May.
Mercer University School of Engineering
engineering.mercer.edu/ms-tcm/

Mark Your Calendar

Organization Events Across the Globe

F.Y.I. lists information about nonprofit ventures only. Please send information to *intercom@stc.org.*

1 19-23 Sept
The **Human Factors and Ergonomics Society** will hold its Annual Meeting 19-23 September at the Washington Hilton, Washington, DC.
HFES
https://hfes.org/web/HFESMeetings/2016annual-meeting.html

2 2-5 Oct
The **IEEE Professional Communication Society** will hold its annual conference, ProComm 2016, 2-5 October at the University of Texas, Austin, TX.
IEEE Professional Communication Society
http://sites.ieee.org/pcs/future-locations/

3 5-8 Oct
The **American Medical Writers Association (AMWA)** will hold its annual Medical Writing & Communication Conference, 5-8 October at the Sheraton Denver Downtown Hotel, Denver, CO.
AMWA
http://www.amwa.org/content.asp?contentid=74
240-238-0940

4 6-8 Oct
The **Council for Programs in Technical and Scientific Communication (CPTSC)** will hold its Annual Meeting 6-8 October at the Georgia Conference Center, Savannah, GA. The conference theme is (Re) Considering Programs in Terms of Methods, Methodologies, and Practices.
CPTSC
http://writeprofessionally.org/cptsc2016/
meloncon@tek-ritr.com

5 14-18 Oct
The **Association for Information Science and Technology** will hold its annual meeting, 14-18 October in Copenhagen, Denmark.
ASIS&T
https://www.asist.org/events/annual-meeting/annual-meeting-2016/
301-495-0900

6 23-25 Oct
The **Public Relations Society of America (PRSA)** will hold its 2016 International Conference, 23-25 October at the JW Mariott in Indianapolis, IN.
PRSA
https://www.prsa.org/Conferences/International-Conference/index.html

7 2-5 Nov
The **American Translators Association (ATA)** will hold its 57th Annual Conference, 2-5 November, at the Hyatt Regency in San Francisco, CA.
ATA
http://www.atanet.org/conf/2016/
+1-703-683-6100
ata@atanet.org

8 23-26 Jan 2017
The **Annual Reliability and Maintainability Symposium** will be held 23-26 January 2017 at the Rosen Plaza Hotel in Orlando, FL.
RAMS
http://www.rams.org/
RAMS2017@rams.org

* STC-related event

Do You Want to See My Monkey?

BY BRENDA HUETTNER | *STC Fellow*

MAGIC CARPET GOLF in Tucson, Arizona, was a local landmark for over 40 years. There were two 18-hole courses, each dotted with an amazing variety of concrete sculptures, including a giant bull, a Buddha, a serpent, an ostrich, some aliens, and a tyrannosaurus. There was a 9,000-pound alien named Goop, a concrete outhouse, and lots of gossip about shenanigans that might have gone on inside the two-story-tall tiki head. When the property was purchased by a car dealership, the sculptures were auctioned off with the condition that the new owners had to move them pronto to make room for a parking lot.

But the piece that caught my eye—the first sculpture visible from the street—was the giant monkey hanging from a concrete palm tree. Approximately 14 feet to the top of the tree, this monkey has eyes that light up and a tail that swings back and forth, designed to knock your golf ball off track. So I placed a very low-ball bid, and to my horror, had actually entered the highest bid. The monkey was mine!

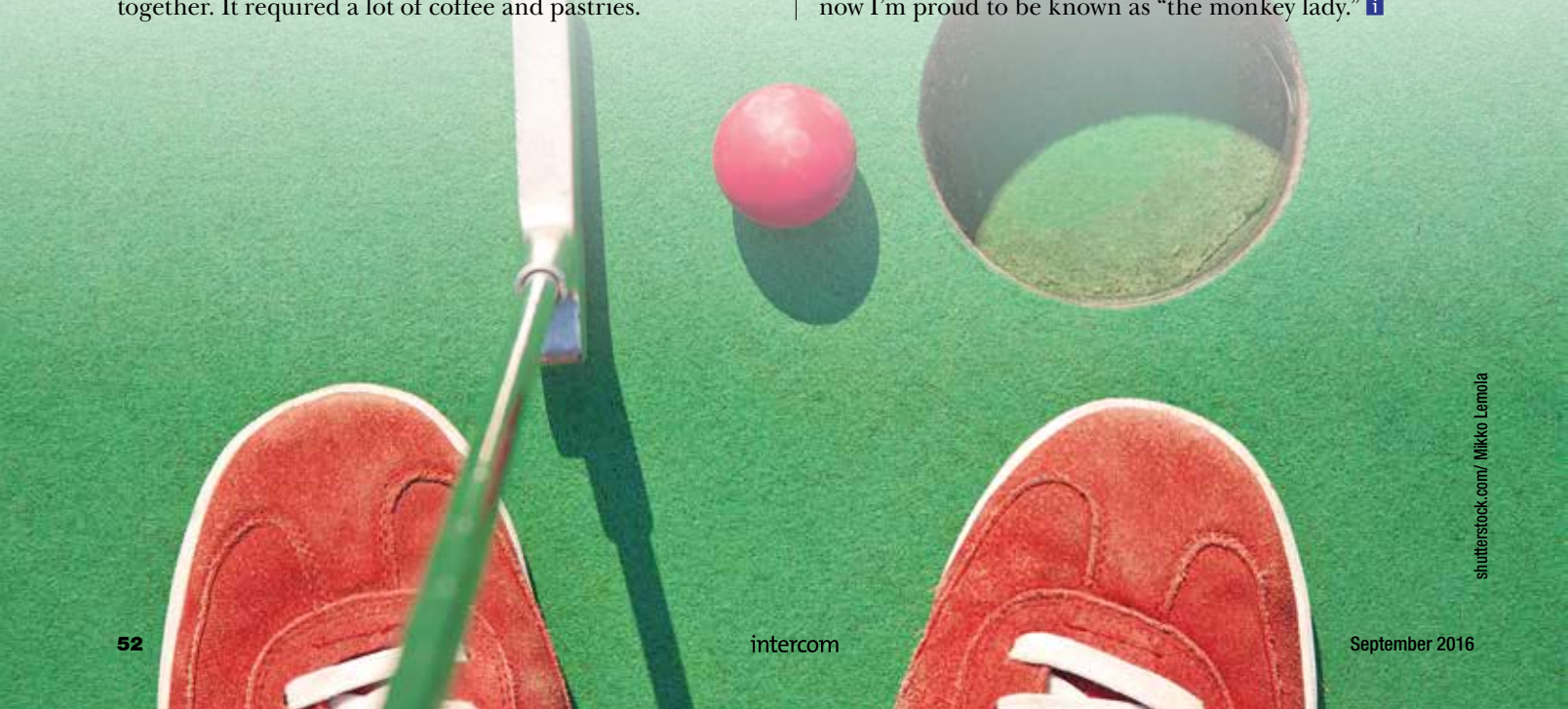
The challenge was in moving him. It turns out, a sculpture that big, one that had been sculpted in place, is not easy to move. Not only did we have to dig him out of the ground (requiring a special giant concrete saw), we also had to cut him into pieces. We had a whole team of people with straps and boards and other supports, a huge crane, people from traffic control, media, and lots of onlookers.

But moving him wasn't the hard part. Once the pieces were safely in my yard (right near downtown Tucson), we still had to dig footings into the hard-packed caliche, get the crane back to lift the pieces, and then convince someone to get up on a ladder and weld everything back together. It required a lot of coffee and pastries.

Once the statue was standing up, a plasterer came to make repairs, and we sandblasted all the old paint off of him. A neighborhood artist painted him to look as natural as possible in his new home.

At one point, a woman I didn't know came up to me and poked me with her finger. "Is that your monkey?" she asked. I couldn't quite tell if she was pleased with this incoming neighbor or not, but I confessed that it was, indeed, my monkey. She gave me a big hug and said, "Thank you so much for saving a piece of Tucson history."

I didn't start out having a "thing" for monkeys, the way some people really like penguins or alligators or cats. But now I'm proud to be known as "the monkey lady." i



shutterstock.com/ Mikko Lemola

STC SUMMIT 2017

SAVE THE DATE

WASHINGTON, DC IN NATIONAL HARBOR, MD

7–10 MAY

TECHNICAL COMMUNICATION SUMMIT '17

STC's 64th Annual Conference